

# A Slot-based Real-time Scheduling Algorithm for Concurrent Transactions in NoC

Bach D. Bui, Rodolfo Pellizzoni, Marco Caccamo  
University of Illinois at Urbana-Champaign  
Department of Computer Science  
{bachbui2, rpelliz2, mcaccamo}@uiuc.edu

## Abstract

We address the problem of scheduling real-time transactions in Network-on-Chip (NoC). In particular, we propose a novel slot-based scheduling algorithm for acyclic transaction sets in NoC. The algorithm induces a competitive sufficient schedulability utilization bound. Since the proposed algorithm is able to exploit the parallelism between non-overlapping transactions, under given assumptions, it performs better than the existing fixed-priority solutions. We evaluate performance through extensive simulations. Furthermore, we discuss some important factors in the implementation of the algorithm and present an implementation in a real system. The measurement shows that the proposed algorithm has relatively low overhead.

## 1 Introduction

Demands for high performance computing systems have recently created significant interest in many-core System-on-Chips (SoC) both industry-wise and academic-wise [10, 11, 6]. In a many-core SoC, communication between processing elements which are interconnected by a Network-on-Chip (NoC) [10, 11, 12] can greatly affect the performance of the system as a whole. Due to their multiple degrees of parallelism both in computation and communication, using SoC in real-time systems requires new theoretical frameworks. In this paper, we address the problem of real-time communication in NoC.

Scheduling of real-time transactions in a NoC while maximizing its utilization poses new challenges which are different from those of traditional real-time systems. In particular, although the problem has the form of multiple resource scheduling, it is not the same as multi-processor scheduling because a transaction may use multiple resources, i.e. physical links, simultaneously. To the best of our knowledge, this scheduling problem has not received adequate attention. In the recent related works [21, 20], Shi et al. proposed a method to calculate worst-case latency of transactions scheduled by a *fixed-priority* scheduling algorithm. The proposed method does not take advantage of the parallelism available between non-overlapping transactions when they share links with a lower-priority one.

To tackle this problem, we advocate the use of *slot-based* scheduling, in which time line is divided into consecutive equal slots and transactions are scheduled on contention-free slots. This type of scheduling has been used to build PFair [4] and BF [25], which are the optimal scheduling algorithms for multi-processors. Although this approach requires synchronization between bus el-

ements and computation at the end node (i.e. bus elements), it can significantly reduce implementation complexity of real-time NoC because it eliminates the need of buffers and arbiters at the routers. The slot-based scheduling model has been successfully implemented in the Aethereal NoC [12], a guaranteed-service NoC developed at Phillips Laboratory.

In [9], Bui et al. proposed a slot-based scheduling algorithm for multi-domain uni-dimensional buses. These are buses whose routers are arranged on a straight line. Although the algorithm has a very competitive performance compared to other solutions, it *only* works on uni-dimensional NoC. Due to the complexity of the addressed problem, in this research, we will focus on acyclic transaction sets and propose a scheduling algorithm for these sets in multi-dimensional NoC architectures. As shown later, this type of transaction sets appear in many practical applications. Although, the proposed algorithm uses some background concepts from [9], it requires a new theoretical framework compared to [9]. Since the proposed algorithm is able to exploit the parallelism between non-overlapping transactions, our extensive simulated experiments show that it performs better than the fixed-priority ones on acyclic transaction sets. Our online implementation of the proposed algorithm in a real system also shows that it has relatively low overhead.

Our paper is organized as follows. We review related works in Section 2. The detailed scheduling problem is discussed in Section 3 and the algorithms are proposed in Section 4. We discuss important implementation issues and the measurement of the algorithm's overhead in Section 5. Finally, we evaluate the proposed solution through extensive simulation in Section 6.

## 2 Related Works

Existing works on hard real-time communication [15, 22, 19] focus on communication between computers on *single-domain* bus networks. In these networks, only one transaction can be transferred on a bus at any time because the bus is shared between all transactions (thus named "single-domain"). A system with multiple buses is considered in [13]. However, each bus in the system still has one domain. Since a single-domain bus bears a similarity to single-processor systems, the traditional real-time scheduling theory for single-processor systems [17] is applied or extended to solve the addressed problem in these works. The NoC of many-core SoCs in which we are interested have multi-domain buses where non-overlapping transactions can be transferred concurrently. In addition, the number of domains on a bus is determined by the topology of bus transactions.

There has also been significant research into real-time communication in NoC. Most of these works [20, 21, 3, 16, 24] are con-

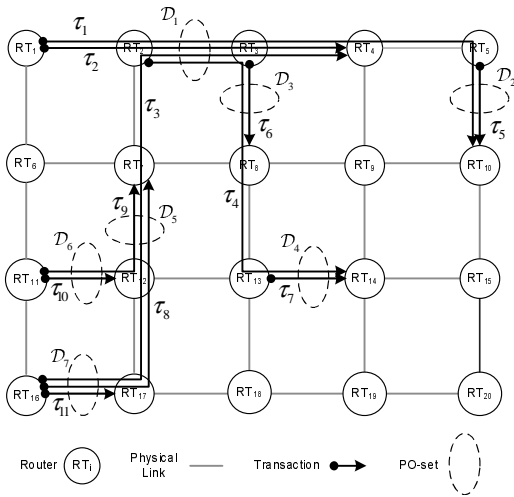


Figure 1. A transaction set in NoC

cerned with the fixed-priority scheduling paradigm. For example, in [20, 21], Zheng et al. propose a solution to optimally assign fixed priorities to real-time transactions and a method to analyze the worst-case transaction latency (WTL) under a fixed-priority scheduling algorithm. Unlike these works, our proposed algorithm schedules transactions dynamically based on the graphical relationship between them.

In [9], Bui et al. proposed a slot-based scheduling algorithm for uni-dimensional NoCs. This algorithm, however, cannot be applied to many-dimensional NoCs. The reason is that the algorithms in [9] rely on the fact that for each transaction set on ring buses there exists a *total order* which is defined solely based on transactions' endpoints. The total order exists because a ring bus can be represented as a straight line (i.e. uni-dimensional). This total order does not exist in a general NoC. Furthermore, in this paper, we will propose an algorithm which has *polynomial* time complexity while that of [9] has *pseudo polynomial* time complexity. Finally, to evaluate the performance of the proposed algorithm, this paper also presents the results of extensive experiments and an implementation of the algorithm in a real system, which [9] does not have.

### 3 Real-time Bus Transaction and Scheduling Model

We consider a system-on-chip comprising multiple Processing Elements (PEs) interconnected by a NoC. The NoC is composed of routers connected by communication links. Links can be either unidirectional or bidirectional, but each link is single-duplex which supports only a *single* data transmission at a time. Most NoC employ full-duplex connections; in our model, they are represented by two unidirectional links. We do not impose any restriction on network topology: mesh, torus, ring, tree and irregular topologies can all be supported. Figure 1 shows an example of a  $4 \times 5$  mesh NoC.

Applications running on PE request periodic data transfers, called *data transactions*. A data transaction comprises an infinite number of periodic jobs. We define  $\mathcal{T}$  as the set of all data transactions:  $\mathcal{T} = \{\tau_i : i = [1, N]\}$ . A data transaction  $\tau_i$  is characterized by a tuple  $\tau_i = (e_i, p_i, \mathcal{R}_i)$  where  $e_i$  is the time required to transfer each job of  $\tau_i$ ;  $p_i$  is the period of  $\tau_i$ ; and  $\mathcal{R}_i$  is the fixed route of  $\tau_i$  through the network, that is the set of links that the transaction traverses. As an example, the route of transaction  $\tau_4$  in Figure 1 is  $\mathcal{R}_4 = \{RT_2 \rightarrow RT_3 \rightarrow RT_8 \rightarrow RT_{13} \rightarrow RT_{14}\}$ , where  $\rightarrow$  rep-

resents a physical link. We assume that when  $\tau_i$  is transmitted, it uses all links in  $\mathcal{R}_i$  at the same time. This is the assumption used in NoC with wormhole switching [10, 11], which are the most popular switching protocol used in NoC. Note that this assumption does not imply that the link resource is wasted because, in practice, a packet will be split into multiple flits, which then will be sent in pipeline along its route. Each job of  $\tau_i$  must complete within its period, i.e. relative deadlines are equal to periods. The network utilization  $u_i$  of  $\tau_i$  is calculated as:  $u_i = e_i/p_i$ . We assume that all data transactions arrive at time 0. Let hyper-period  $h$  of  $\mathcal{T}$  be the least common multiple of the periods of all transactions in  $\mathcal{T}$ . Finally, two transactions  $\tau_i$  and  $\tau_j$  are said to *overlap* if their routes have any link in common, that is  $\mathcal{R}_i \cap \mathcal{R}_j \neq \emptyset$ . Given a data transaction set  $\mathcal{T}$ , we define an overlap indicating function  $OV : \mathcal{T} \times \mathcal{T} \mapsto \{0, 1\}$  where  $OV(\tau_i, \tau_j) = 1$  if  $\tau_i$  and  $\tau_j$  overlap, and 0 otherwise.

**Scheduling model:** We adopt a *slot-based, contention-free* scheduling model<sup>1</sup>. Time is divided into slots of fixed size, indexed starting from 0. For each transaction  $\tau_i$ , both its transfer time  $e_i$  and period  $p_i$  are expressed in terms of the basic slot size. Let  $a$  be the minimum amount of data in bytes that can be completely transmitted between any two processing elements in the system within a single time slot, assuming no other data transfer is performed on the NoC. Then if  $\tau_i$  must transmit  $b_i$  bytes of data every period, its transfer time can be computed as  $e_i = \lceil \frac{b_i}{a} \rceil$ . Note that, in NoC, the transmitting latency of a packet increases with the length of its route. However, in reality, this increase in latency is relatively small compared to the total latency. As an example in the Cell processor [2], the total latency of a smallest-size packet is 80 cycles and each more hop in the packet's route adds only 1 cycle to the total latency.

A schedule  $S$  is defined as a function  $S : \mathcal{T} \times \mathbb{N} \mapsto \{0, 1\}$  where  $S(\tau_i, t) = 1$  if and only if  $\tau_i$  is scheduled to be transmitted during slot  $t$ . A schedule  $S$  is *valid* if and only if it is contention-free, that is according to  $S$ , no two overlapping transactions are scheduled in the same slot.  $S$  is a *feasible* schedule of  $\mathcal{T}$  if it is valid and according to  $S$ , every job of every transaction finishes before its deadline.

**PO-sets and incident tree:** A *pairwise overlap set* (PO-set)  $\mathcal{D}_j$  is defined as a maximal set of overlapping transactions, i.e. a subset of  $\mathcal{T}$  that satisfies the following two conditions: 1)  $\forall \tau_i, \tau_j \in \mathcal{D} : OV(\tau_i, \tau_j) = 1$ ; and 2) if  $\tau_k \notin \mathcal{D}$  then  $\exists \tau_i \in \mathcal{D} : OV(\tau_i, \tau_k) = 0$ . By definition, every two transactions that overlap each other must belong to at least a same PO-set. For convenience, we consider that a non-overlapping transaction belongs to a PO-set that contains only that transaction. Let  $N^{\mathcal{D}}$  be the total number of PO-sets of  $\mathcal{T}$ . In general a transaction may belong to more than one PO-set. As an example, the transaction set in Figure 1 comprises seven PO-sets:  $\mathcal{D}_1 = \{\tau_1, \tau_2, \tau_3, \tau_4\}$ ,  $\mathcal{D}_2 = \{\tau_1, \tau_5\}$ ,  $\mathcal{D}_3 = \{\tau_4, \tau_6\}$ ,  $\mathcal{D}_4 = \{\tau_4, \tau_7\}$ ,  $\mathcal{D}_5 = \{\tau_3, \tau_8, \tau_9\}$ ,  $\mathcal{D}_6 = \{\tau_9, \tau_{10}\}$ ,  $\mathcal{D}_7 = \{\tau_3, \tau_8, \tau_{11}\}$ . Two PO-sets  $\mathcal{D}_i, \mathcal{D}_j$  are said to be *connected* if  $\mathcal{D}_i \cap \mathcal{D}_j \neq \emptyset$ .

The *PO-graph* of  $\mathcal{T}$  is defined as the incident graph of PO-sets of  $\mathcal{T}$ , i.e. the graph whose vertexes represent PO-sets and there is an edge between two vertexes if the two correspondent PO-sets are connected. Thereafter we shall use a PO-set and its correspondent vertex in PO-graph interchangeably. We assume that a PO-graph is a connected graph. The reason is that if the graph is disconnected, its components do not interfere with each other in terms of scheduling and therefore we can deal with them as separate transaction sets. Figure 2(a) shows the PO-graph of the transaction set showed in Figure 1.

<sup>1</sup>A hardware implementation of this model is presented in [12].

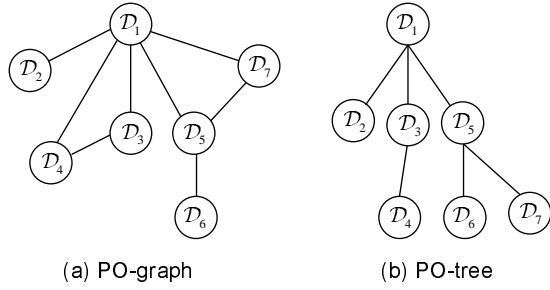


Figure 2. PO-graph, PO-tree of  $\mathcal{T}$  shown in Figure 1

Let *PO-tree* denote a spanning tree of the PO-graph and let PO-set  $\mathcal{D}^r$  be the root of the PO-tree. Figure 2(b) shows a PO-tree rooted at  $\mathcal{D}_1$  of the PO-graph in Figure 2(a). With respect to a specific PO-tree, PO-set  $\mathcal{D}_i$  is an *ancestor* of PO-set  $\mathcal{D}_j$ , denoted by  $\mathcal{D}_i \prec \mathcal{D}_j$ , if  $\mathcal{D}_i$  is on the path from the root  $\mathcal{D}^r$  to  $\mathcal{D}_j$  on the PO-tree. Let  $\text{ancestors}(\mathcal{D}_j)$  be the set of all ancestors of  $\mathcal{D}_j$ . For convenience, we use notation  $\mathcal{D}_i \preceq \mathcal{D}_j$  to indicate that  $\mathcal{D}_i \in \text{ancestors}(\mathcal{D}_j) \cup \mathcal{D}_j$ . Furthermore,  $\mathcal{D}_i$  is the *parent* of  $\mathcal{D}_j$ , denoted by  $\mathcal{D}_i = \text{parent}(\mathcal{D}_j)$ , if and only if  $\mathcal{D}_i$  is the immediate ancestor of  $\mathcal{D}_j$ . Note that in a tree, a non-root node has a unique parent. Let  $\text{children}(\mathcal{D}_i)$  be the set of all PO-sets whose parent is  $\mathcal{D}_i$ . As an example, in Figure 2(b),  $\text{ancestors}(\mathcal{D}_6) = \{\mathcal{D}_1, \mathcal{D}_5\}$ ,  $\mathcal{D}_5 = \text{parent}(\mathcal{D}_6)$ , and  $\text{children}(\mathcal{D}_5) = \{\mathcal{D}_6, \mathcal{D}_7\}$ . We define the height of tree node  $\mathcal{D}_i$ , denoted by  $\text{height}(\mathcal{D}_i)$ , as follows: if  $\text{children}(\mathcal{D}_i) = \emptyset$  then  $\text{height}(\mathcal{D}_i) = 0$ , otherwise  $\text{height}(\mathcal{D}_i) = 1 + \max_{\mathcal{D}_j \in \text{children}(\mathcal{D}_i)} \text{height}(\mathcal{D}_j)$ .

In [7], we proved that there exists a category of transaction sets in uni-dimensional buses, called *cyclic* transaction sets, for which the scheduling problem is NP-complete. Since uni-dimensional buses are a special case of NoC, the same result applies in our case. In this paper, we extend the definition of acyclic/cyclic transaction sets in [9, 7] to encompass transaction sets on multi-dimensional NoC. We say that a NoC transaction set is acyclic if it has a PO-tree that satisfies Property 3.1 and 3.2, and it is cyclic otherwise.

**Property 3.1** If  $\mathcal{D}_l \preceq \mathcal{D}_m \preceq \mathcal{D}_n$  and  $\tau_i \in \mathcal{D}_l$  and  $\tau_i \in \mathcal{D}_n$ , then  $\tau_i \in \mathcal{D}_m$ .

**Property 3.2** If  $\mathcal{D}_l \not\preceq \mathcal{D}_m$  and  $\mathcal{D}_m \not\preceq \mathcal{D}_l$ , then  $\mathcal{D}_l \cap \mathcal{D}_m = \emptyset$ .

Note that a transaction set that is cyclic or acyclic by the definition in [9, 7] is also cyclic or acyclic, respectively, by the current definition (see proof in Appendix). In this research, we will focus only on acyclic transaction sets because of two reasons: 1) as we discuss later, many real-time applications exhibit data-flow topologies that are acyclic; 2) for this instance of the problem, we can derive an efficient and near-optimum solution in practical settings. We believe that having a good solution for this specific problem will provide a good theoretical foundation for solving the general NP-complete problem. Thereafter, we use PO-tree to denote the spanning tree that satisfies the aforementioned properties. Note that if  $\mathcal{T}$  is acyclic, then its number of PO-sets  $N^{\mathcal{D}} \leq N$  (see proof in Appendix). The transaction set shown in Figure 1 is acyclic because given its PO-tree shown in Figure 2(b), all transactions and PO-sets satisfy Property 3.1 and 3.2. For example, since  $\mathcal{D}_1 \prec \mathcal{D}_3 \prec \mathcal{D}_4$ , any transaction that is in both  $\mathcal{D}_1$  and  $\mathcal{D}_4$  (e.g.  $\tau_4$ ) is also in  $\mathcal{D}_3$  (Property 3.1), or since  $\mathcal{D}_2 \not\preceq \mathcal{D}_3$  and  $\mathcal{D}_3 \not\preceq \mathcal{D}_2$ ,  $\mathcal{D}_2 \cap \mathcal{D}_3 = \emptyset$  (Property 3.2).

For ease of presentation, we define the following index scheme for the PO-sets in the PO-tree: select any Depth-First-Search (DFS)

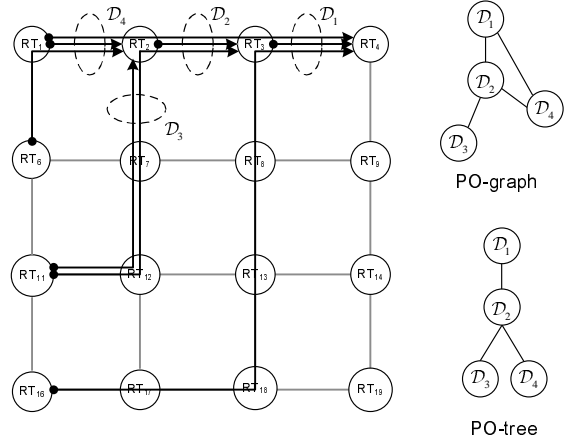


Figure 3. A data-stream processing application

travel over PO-tree. Then each PO-set is indexed by a unique number from 1 to  $N^{\mathcal{D}}$  in the order it is visited in the selected DFS. PO-sets in Figure 2(b) are indexed following this definition. Note that, with this index scheme, if  $\mathcal{D}_l \preceq \mathcal{D}_m$  then  $l \leq m$  but  $l \leq m$  does not always imply  $\mathcal{D}_l \preceq \mathcal{D}_m$  (see  $\mathcal{D}_2$  and  $\mathcal{D}_3$  in Figure 2). The following property, however, is true (see proof in Appendix).

**Property 3.3** If  $l \leq m \leq n$  and  $\mathcal{D}_l \preceq \mathcal{D}_n$ , then  $\mathcal{D}_l \preceq \mathcal{D}_m$ .

Given an indexed PO-sets, let  $\text{maxid}_i$  and  $\text{minid}_i$  denote the indexes with maximum and minimum value, respectively, among all PO-sets to which  $\tau_i$  belongs. Also denote these PO-sets as  $\text{maxPO}_i$  and  $\text{minPO}_i$ , respectively. As an example,  $\text{maxid}_3 = 7$ ,  $\text{minid}_3 = 1$ ,  $\text{maxPO}_3 = \mathcal{D}_7$ ,  $\text{minPO}_3 = \mathcal{D}_1$ . We have the following property (see proof in Appendix).

**Property 3.4** For every  $\mathcal{D}_l$  which contains  $\tau_i$ ,  $\text{minPO}_i \preceq \mathcal{D}_l$ .

Since by definition, no two transactions of a PO-set  $\mathcal{D}_l$  can be scheduled concurrently, all transactions of  $\mathcal{D}_l$  must be scheduled in sequence. In other words, the transactions of  $\mathcal{D}_l$  can be considered to be sharing one resource. This results in a necessary condition on the schedulability of a transaction set shown in Theorem 3.1

**Theorem 3.1** A transaction set  $\mathcal{T}$  is schedulable only if:

$$\forall \mathcal{D}_l \subset \mathcal{T} : u_l^{\mathcal{D}} = \sum_{\tau_i \in \mathcal{D}_l} u_i \leq 1. \quad (3.1)$$

**Motivating Applications:** Many real-time applications are in the form of data-flow processing applications [18, 23] where data may be processed through multiple consecutive stages. An example is the multipurpose status display application on an avionic system [18] which shows the status of all aircraft avionics devices. A task periodically gets data from I/O devices such as radars, then processes the data before sending information to a display task which is in charge of displaying useful information to the pilots. Another example is an application that processes transmission flows in 3G base stations [23]. In the down-link direction, the input flows are the data flow and the control flow. These flows are processed separately through several stages then merged back into one flow. The merged flow then goes through several additional processing stages before being sent to the output port. A popular programming model for this type of applications in SoC is the streaming model [1, 23] in which each processing stage is executed in one PE. PEs

are in either a serial or parallel pipeline. Data is transferred between processing stages through the NoC. If the streaming programming model is used, it is usually easy to come up with an allocation of stages to PEs that result in acyclic transactions. Figure 3 shows an acyclic transaction set created by a data-flow processing applications: input flows are processed in parallel by processing elements at  $\{RT_1, RT_6\}$ ,  $\{RT_{11}, RT_{16}\}$ , then merged at  $RT_2$  and processed by the processing element at this router; the merged flow continues going through two additional processing stages at  $RT_3$  and  $RT_4$ . The induced PO-graph has the following PO-set nodes:  $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_4$ . In Figure 3, we represent each PO-set with a dashed circle where transactions that cut through a circle are transactions in the correspondent PO-set of the circle. The PO-graph and a PO-tree (rooted at  $\mathcal{D}_1$ ) of this transaction set is also shown in Figure 3. The transaction set is acyclic because it satisfies Property 3.1 and 3.2. For example: the transaction which is in both  $\mathcal{D}_1$  and  $\mathcal{D}_4$  is also in  $\mathcal{D}_2$  (Property 3.1); and  $\mathcal{D}_3 \cap \mathcal{D}_4 = \emptyset$  (Property 3.2).

## 4 Scheduling Algorithms

Our work in [9] has introduced two scheduling algorithms for real-time transactions on a ring bus: Algorithm POBase is designed to schedule transactions whose periods are the same, whereas Algorithm POGen is designed for general transaction sets. The latter, which is summarized in Section 4.1, is built upon two algorithms POBase and GenerateLoad. Although the general framework of POGen can be used for transaction sets on NoC, Algorithm POBase and GenerateLoad proposed in [9] can not be reused. The reason is that the two algorithms in [9] rely on the fact that for each transaction set on a ring bus, there exists a *total order* which is defined solely based on transactions' endpoints. This total order does not exist in a general NoC. Therefore, a different transaction order (more general) must be defined. Moreover, the new proposed algorithms have polynomial time complexity which are  $O(N^2 \log(N))$  and  $O(N^2)$  respectively, while algorithms proposed in [9] have pseudo-polynomial time complexity.

### 4.1 Background

In this section we will summarize Algorithm POGen proposed in [9]. The online version of POGen is shown in Algorithm 1. To generate the schedule, we divide the execution time-line into consecutive *scheduling intervals*. Each scheduling interval, denoted by  $\text{int}^k$ , is defined as a set of slots between two closest arrival times (also deadlines) of transactions in the transaction set. Let  $t^k$  and  $t^{k+1}$  be the beginning and end of  $\text{int}^k$ , respectively. Denote size of  $\text{int}^k$  as  $|\text{int}^k|$ . In scheduling interval  $\text{int}^k$ , transaction  $\tau_i$  will be executed in  $l_i^k$  slots where  $l_i^k$  is called the *interval load* of  $\tau_i$  in  $\text{int}^k$ . POGen is invoked at the beginning of each scheduling interval and generates the schedule for that interval in two steps. First, it uses function GenerateLoad to generate an interval load for every transaction. Then, given the generated interval loads, it uses function POBase to generate the schedule in the interval for every transaction.

With regard to interval loads, we define for each transaction  $\tau_i$  and scheduling interval  $\text{int}^k$  a lag function:

$$\text{lag}(\tau_i, \text{int}^k) = u_i * t^{k+1} - \sum_{x \in [0, t^k]} S(\tau_i, x).$$

The function calculates how much time  $\tau_i$  must be executed in interval  $\text{int}^k$  such that at the end of  $\text{int}^k$  it is scheduled according to the fluid scheduling model. We also define for each PO-set  $\mathcal{D}_j$  a similar lag function:

$$\text{lag}(\mathcal{D}_j, \text{int}^k) = u_j^{\mathcal{D}} * t^{k+1} - \sum_{\tau_i \in \mathcal{D}_j} \sum_{x \in [0, t^k]} S(\tau_i, x).$$

Let the set of interval loads in scheduling interval  $\text{int}^k$  that satisfies Inequality 4.1 and 4.2 be a *feasible load set* of  $\text{int}^k$ .

$$\forall \tau_i \in \mathcal{T} : \lfloor \text{lag}(\tau_i, \text{int}^k) \rfloor \leq l_i^k \leq \lceil \text{lag}(\tau_i, \text{int}^k) \rceil, \quad (4.1)$$

$$\begin{aligned} \forall \mathcal{D}_j \subset \mathcal{T} : \lfloor \text{lag}(\mathcal{D}_j, \text{int}^k) \rfloor &\leq \sum_{\tau_i \in \mathcal{D}_j} l_i^k \\ &\leq \min(|\text{int}^k|, \lceil \text{lag}(\mathcal{D}_j, \text{int}^k) \rceil). \end{aligned} \quad (4.2)$$

---

#### Algorithm 1 POGen

---

**Input:** transaction set  $\mathcal{T}$ , interval  $\text{int}^k$ ,  $S$  for intervals before  $t^k$

**Output:** schedule  $S$  for interval  $\text{int}^k$

- 1:  $\{l_i^k : \forall i \in [1, N]\} \leftarrow \text{GenerateLoad}(\mathcal{T}, \text{int}^k, S)$
  - 2:  $\mathcal{T}' \leftarrow \{\{l_i^k, |\text{int}^k|, \mathcal{R}_i\} : \forall i \in [1, N]\}$
  - 3:  $S$  for interval  $\text{int}^k \leftarrow \text{POBase}(\mathcal{T}')$
- 

Inequality 4.1 sets conditions on the interval load for each transaction, based on the closest integral values of the lag functions. Inequality 4.2 sets conditions on the total interval load of each PO-set.

We proved in [9] that the schedule generated by executing POGen iteratively over all scheduling intervals is feasible if the following two conditions are true.

**Condition 4.1** POBase is optimal for a transaction set whose transactions' periods are the same, meaning that the schedule generated by POBase is feasible if the transaction set satisfies the necessary condition in Theorem 3.1.

**Condition 4.2** GenerateLoad generates a feasible load set  $\{l_i^k : \forall \tau_i \in \mathcal{T}\}$  for interval  $\text{int}^k$  if its inputs satisfy the following Inequalities:

$$\forall \mathcal{D}_j \subset \mathcal{T} : \lfloor \text{lag}(\mathcal{D}_j, \text{int}^k) \rfloor \leq |\text{int}^k| \quad (4.3)$$

$$\sum_{x \in [0, t^k]} S(\tau_i, x) \geq \lfloor u_i * t^k \rfloor \quad (4.4)$$

Note that Inequalities 4.3, 4.4 express conditions on the schedule generated for the previous intervals  $\text{int}^0, \dots, \text{int}^{k-1}$ ; the proof is by induction over the sequence of scheduling intervals starting from  $\text{int}^0$ . Due to space constraints, we refer readers to [9] for detailed description of POGen and the proof. Note that Algorithm 1 is the same as the one in [9], except that the latter is presented as an offline algorithm which generates the schedule for all intervals at once.

In the following sections, we will propose algorithm POBase and GenerateLoad for acyclic transaction sets on NoC that satisfy Conditions 4.1 and 4.2. To differentiate with the algorithms in [9], we name the new algorithms POBaseNoC and GenerateLoadNoC. The proposed POBaseNoC satisfies Condition 4.1 because it feasibly schedules any same-period acyclic NoC transaction set that satisfies the necessary condition of Theorem 3.1. Furthermore, the

proposed GenerateLoadNoC satisfies Condition 4.2 if all PO-sets satisfy the following utilization bound:

$$\forall \mathcal{D}_j \subset \mathcal{T} : u_j^{\mathcal{D}} \leq \frac{L-1}{L}, \quad (4.5)$$

where  $L$  is defined as the greatest common divisor (GCD) of all transaction periods and is measured in terms of the basic slot size. Although the utilization bound is sufficient, it approximates 1 when  $L$  is large. The value of  $L$  in a real system is a function of both the slot size and transaction periods. As we discuss in more details in Section 5, we believe that in many modern NoCs  $L$  has practical values ranging from 10 to 100 slots. Since the proposed GenerateLoadNoC imposes a stricter requirement on the transaction set than that of POBaseNoC, Inequality 4.5 becomes a sufficient utilization bound of POGen.

## 4.2 The POBaseNoC algorithm

The goal of POBaseNoC is to find a feasible schedule for a transaction set which satisfies necessary condition in Theorem 3.1 and has all transactions with a same period  $p$ . The proposed POBaseNoC is executed as follows: each transaction, in ascending order of its *minid*, is assigned to the earliest *valid* slots (transactions with the same *minid* are ordered by their indexes). Slot  $t$  is valid for transaction  $\tau_i$  if  $t$  has not been assigned to transactions overlapping with  $\tau_i$ . In the next paragraphs, we will discuss an efficient implementation of POBaseNoC and prove its correctness and optimality.

In POBaseNoC, in order to find valid slots for a transaction, we scans through a list of durations (Line 5) (instead of scanning through all slots as in [9]). Each duration represents a set of consecutive slots to which some non-overlapping transactions have been assigned. A transaction  $\tau_i$  can only be assigned to slots in duration  $d$  if transactions that have been assigned to  $d$  do not overlap with  $\tau_i$ . We call these durations the *valid* durations of  $\tau_i$ .

In POBaseNoC,  $Dur$  stores the list of durations. Each duration  $d$  is represented as a tuple  $\{d.t_1, d.t_2, d.tlist\}$  where all and only slots in  $[d.t_1, d.t_2)$  are slots of  $d$ . The size of  $d$  is the number of slots of  $d$ .  $d.tlist$  is the list of transactions that have been assigned to  $d$ . Transaction  $\tau_i$  is said to be *assigned* to  $d$  if and only if  $\tau_i$  is assigned to all slots in  $d$ . The algorithm guarantees that a transaction is assigned to all slots of a duration or none of them. The first duration appended to  $Dur$  is  $\{0, p, \{\text{null}\}\}$  where  $\{\text{null}\}$  denotes an empty list. A duration will be updated or split into two durations after an assignment of a transaction completes.  $Sched(\tau_i)$  stores the list of pairs  $\{t_1, t_2\}$  where all slots in  $[t_1, t_2)$  are assigned to  $\tau_i$ , or equivalently  $\forall t \in [t_1, t_2) : S(\tau_i, t) = 1$ . Consider when transaction  $\tau_i$  is being assigned. The value of variable  $r$  will be the remaining amount of transmission time of  $\tau_i$  to be assigned. The algorithm checks if duration  $d$  is valid for  $\tau_i$  using Line 6 to 9 (we will describe these steps in next paragraphs). If  $d$  is valid then: 1) if the remaining transmission time  $r$  of  $\tau_i$  is larger than the size of  $d$ ,  $\tau_i$  is assigned to all slots of  $d$  (Line 12). In this case,  $d$  will be updated by adding  $\tau_i$  to  $d.tlist$  (Line 11); 2) otherwise,  $d$  is split into two durations which are  $d_1 = \{d.t_1, d.t_1 + r, d.tlist \cup \{\tau_i\}\}$  and  $d_2 = \{d.t_1 + r, d.t_2, d.tlist\}$  (Line 15), and  $\tau_i$  is assigned to  $d_1$  (Line 16).

---

### Algorithm 2 POBaseNoC

---

**Input:**  $\mathcal{T}$  whose all transactions have same period  $p$

```

1:  $\mathcal{L} \leftarrow$  list of  $\forall \tau_i \in \mathcal{T}$  in ascending order of  $\text{minid}_i$ 
2: add  $d = \{0, p, \{\text{null}\}\}$  to  $Dur$ 
3: for each  $\tau_i \in \mathcal{L}$  do
4:    $r \leftarrow e_i$ 
5:   for each  $d \in Dur$  do
6:      $\mathcal{M} \leftarrow \{\tau_l \in d.tlist : \text{maxid}_l \geq \text{minid}_i\}$ 
7:      $\tau_j \leftarrow \arg \min_{\tau_l \in \mathcal{M}} (\text{maxid}_l)$ 
8:     //check if duration  $d$  is valid for  $\tau_i$ 
9:     if  $\tau_j = \text{None}$  or  $(\tau_j \neq \text{None}$  and  $OV(\tau_j, \tau_i) = 0)$  then
10:      if  $r \geq d.t_2 - d.t_1$  then
11:         $d.tlist \leftarrow d.tlist \cup \{\tau_i\}$ 
12:        add  $\{d.t_1, d.t_2\}$  to  $Sched(\tau_i)$ 
13:         $r \leftarrow r - (d.t_2 - d.t_1)$ 
14:      else
15:        replace  $d \in Dur$  with  $d_1 = \{d.t_1, d.t_1 + r, d.tlist \cup \{\tau_i\}\}$  and  $d_2 = \{d.t_1 + r, d.t_2, d.tlist\}$ 
16:        add  $\{d_1.t_1, d_1.t_2\}$  to  $Sched(\tau_i)$ 
17:         $r \leftarrow 0$ 
18:      break

```

---

Lines 6 to 9 are used to check if  $d$  is valid for  $\tau_i$  (i.e. if there is no transaction in  $d.tlist$  overlapping  $\tau_i$ ). To do this, it suffices to check  $\tau_i$  with only one transaction in  $d.tlist$  which is  $\tau_j = \arg \min_{\tau_l \in \mathcal{M}} (\text{maxid}_l)$  where  $\mathcal{M} = \{\tau_l \in d.tlist : \text{maxid}_l \geq \text{minid}_i\}$  (see Lemma 4.2). This implementation results in a more time-efficient algorithm as will be shown later.

Figure 4 shows a schedule generated by POBaseNoC for the transaction set shown in Figure 1 where the DFS travel is  $\mathcal{D}_1, \mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_4, \mathcal{D}_5, \mathcal{D}_6, \mathcal{D}_7$  and the tuple  $(e_i, p_i)$  of each transaction is given as follows:  $\tau_1 : (2, 8)$ ;  $\tau_2 : (2, 8)$ ;  $\tau_3 : (3, 8)$ ;  $\tau_4 : (1, 8)$ ;  $\tau_5 : (6, 8)$ ;  $\tau_6 : (7, 8)$ ;  $\tau_7 : (7, 8)$ ;  $\tau_8 : (1, 8)$ ;  $\tau_9 : (4, 8)$ ;  $\tau_{10} : (4, 8)$ ;  $\tau_{11} : (4, 8)$ . Given the transactions' parameters, all PO-sets have utilization 100%. The schedule of each PO-set is depicted by a set of continuous rectangles on the horizontal time line. Each rectangle represents an execution of a transaction. A valid transaction ordering in Line 3 is  $\tau_1, \tau_2, \tau_3, \tau_4, \tau_5, \tau_6, \tau_7, \tau_8, \tau_9, \tau_{10}, \tau_{11}$  (i.e. ascending order of transactions' *minid*). Consider when  $\tau_3$  is being allocated. At this point since  $\tau_1$  and  $\tau_2$  have been allocated,  $Dur$  has three durations  $d_1 = \{0, 2, \{\tau_1\}\}$ ,  $d_2 = \{2, 4, \{\tau_2\}\}$ ,  $d_3 = \{4, 8, \{\text{null}\}\}$ . Since only  $d_3$  is valid, POBaseNoC assigns  $\tau_3$  to slots in  $[4, 7)$  and splits  $d_3$  into two durations which are  $\{4, 7, \{\tau_3\}\}$  and  $\{7, 8, \{\text{null}\}\}$ .

The correctness of POBaseNoC is proved in Lemma 4.2 and 4.3 which show that Line 6 to 9 guarantees that every transaction is assigned to only valid durations which contains only valid slots. The proof of these lemmas requires Lemma 4.1. The optimality of the algorithm is, then, shown in Theorem 4.1.

**Lemma 4.1** Consider transaction  $\tau_i$  and  $\tau_j$ . If transaction  $\tau_j$  has been allocated before  $\tau_i$  and  $OV(\tau_i, \tau_j) = 1$ , then  $\tau_j \in \text{minPO}_i$ .

**Proof.**

Since  $OV(\tau_i, \tau_j) = 1$ , there exists  $\mathcal{D}_l$  where  $\tau_i, \tau_j \in \mathcal{D}_l$ . Assume by contradiction that  $\tau_j \notin \text{minPO}_i$ , i.e.  $\mathcal{D}_l \neq \text{minPO}_i$ . Since  $\tau_j$  has been allocated before  $\tau_i$ , by the algorithm operation and the definition of *minid*, we have  $\text{minid}_j \leq \text{minid}_i < l$ . By Property 3.3 and 3.4, we have  $\text{minPO}_j \preceq \text{minPO}_i \prec \mathcal{D}_l$ . Then, by Property 3.1, we have  $\tau_j \in \text{minPO}_i$ .  $\square$

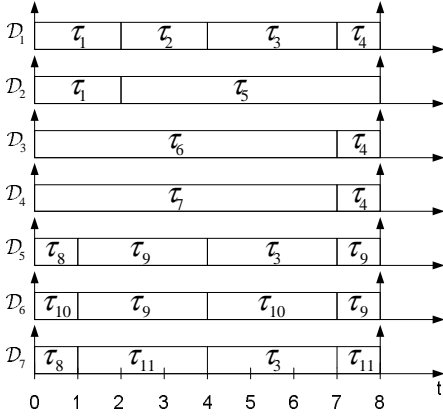


Figure 4. Schedule generated by POBaseNoC

**Lemma 4.2** *If at Line 3, each duration  $d$  has  $d.tlist$  containing only non-overlapping transactions, then transaction  $\tau_i$  will be allocated into only its valid durations.*

**Proof.**

Consider a duration  $d$  and let  $\mathcal{M} = \{\tau_l \in d.tlist : \max id_l \geq \min id_i\}$  and  $\tau_j = \arg \min_{\tau_l \in \mathcal{M}} (\max id_l)$ . We prove this lemma by showing that at Line 6, if there exists transaction  $\tau_k$  in  $d.tlist$  that overlaps  $\tau_i$  (thus  $d$  is invalid) then  $\tau_j \equiv \tau_k$  (thus if-condition at Line 9 is false). Note that  $\tau_k$  does not overlap  $\tau_i$  when  $\max id_k < \min id_i$  because all PO-sets containing  $\tau_k$  are different with that of  $\tau_i$ . Hence, we can assume that  $\tau_k \in \mathcal{M}$ . Assume by contradiction that  $\tau_j \not\equiv \tau_k$ .

We will first prove that the propositions  $A$ ,  $B$ , and  $C$  are true, where  $A \equiv (\min PO_k \preceq \min PO_i)$ ,  $B \equiv (\min PO_j \preceq \min PO_i)$ ,  $C \equiv (\min PO_k \preceq \max PO_j)$ . For  $A$ , since  $\tau_k \in \mathcal{M}$  and  $\tau_k$  has been allocated before  $\tau_i$ , we have  $\min id_k \leq \min id_i \leq \max id_k$ . Then since  $\min PO_k \preceq \max PO_k$  (Property 3.4), by Property 3.3, we have  $A$  is true. We can prove  $B$  true in a similar way by replacing  $\tau_k$  with  $\tau_j$ . For  $C$ , since  $\tau_j \in \mathcal{M}$  and  $\tau_k$  is allocated before  $\tau_i$ , we have  $\min id_k \leq \min id_i \leq \max id_j$ . Together with the definition of  $\tau_j$ , we have  $\min id_k \leq \max id_j \leq \max id_k$ . Then by Property 3.3, we have  $C$  is true.

Due to the tree structure of the PO-tree, from  $A$  and  $B$ , we have either proposition  $D$  or  $E$  is true, where  $D \equiv (\min PO_j \prec \min PO_k)$  and  $E \equiv (\min PO_k \preceq \min PO_j)$ . If  $D$  is true, then together with  $C$ , we have  $F$  is true, where  $F \equiv (\min PO_j \prec \min PO_k \preceq \max PO_j)$ . By Property 3.1,  $F$  implies that  $\tau_j \in \min PO_k$ . This, in turn, implies that  $\tau_j$  overlaps  $\tau_k$  which contradicts the lemma's assumption. If  $E$  is true, then together with  $B$ , we have  $G$  is true, where  $G \equiv (\min PO_k \preceq \min PO_j \preceq \min PO_i)$ . Since  $\tau_k \in \min PO_k$  and  $\tau_k \in \min PO_i$  (Lemma 4.1), by Property 3.1,  $G$  implies that  $\tau_k \in \min PO_j$ . This, in turn, implies that  $\tau_j$  overlaps  $\tau_k$  which, again, contradicts the lemma's assumption. Therefore  $\tau_j \equiv \tau_k$  which means  $d$  is only invalid for  $\tau_i$  when  $\tau_i$  overlaps  $\tau_j$ , otherwise  $d$  is valid and  $\tau_i$  will be allocated into slots of  $d$  (Line 10 to 18).  $\square$

**Lemma 4.3** *Each transaction is allocated into only its valid durations.*

**Proof.**

Since, at the first iteration of Line 3, the assumption of Lemma 4.2 is true (because  $d.tlist = null$ ), by Lemma 4.2 and the operation

from Line 10 to 18, we have  $\tau_1$  is allocated into valid durations and, at the end of this iteration, every duration  $d$  has  $d.tlist$  containing only non-overlapping transactions. Hence, the assumption of Lemma 4.2 is true again at the second iteration of Line 3. The proof, then, is complete by induction reasoning.  $\square$

**Theorem 4.1** *POBaseNoC is optimal for same-period acyclic transaction sets.*

**Proof.**

We will show that if transaction set satisfies the necessary condition in Theorem 3.1, then at the end of the algorithm, the number of valid slots assigned to each transaction  $\tau_i$  is  $e_i$ . Consider when transaction  $\tau_i$  is being allocated. By Lemma 4.1, we have that all transactions whose schedule is allocated before  $\tau_i$  and overlap  $\tau_i$  must belong to  $\min PO_i$ . Therefore invalid slots of  $\tau_i$  must have all been assigned to transactions in  $\min PO_i$ . Assume by contradiction that at the end of the algorithm, the number of valid slots assigned to  $\tau_i$  is smaller than  $e_i$ . Since the algorithm guarantees that a transaction can only be assigned to all slots of a duration or none of them (Line 12 and 16), *all* and *only* valid slots of a transaction are contained in its valid durations. Therefore, the contradiction assumption is true only if at Line 4, the number of valid slots of  $\tau_i$  is smaller than  $e_i$ . This implies that  $p - \sum_{\tau_j \in \min PO_i \setminus \{\tau_i\}} e_j < e_i$ . This contradicts with condition in Theorem 3.1 which implies that  $\sum_{\tau_j \in \min PO_i} e_j \leq p$ . In other words, there are always enough valid slots to allocate  $\tau_i$ . Since this is also true for all other transaction, the proof completes.  $\square$

**POBaseNoC Analysis:** Since a new duration is added only when a transaction assignment is complete i.e.  $r = 0$  (Line 15), the maximum number of durations in  $Dur$  is  $N$ . Therefore, the for-loop between Line 3 and 18 is executed at most  $N^2$  times. Since we can implement  $d.tlist$  as a sorted queue based on  $\max id$ , the operation to look up  $\tau_j$  at Line 6 and 7 will take  $O(\log(N))$  time. And the operation to insert new item into the ordered list  $d.tlist$  at Line 11 and 15 will also take  $O(\log(N))$  time. Furthermore, since the rest of the code between Line 6 and 18 can be implemented with a constant number of operations, the time complexity of POBaseNoC is  $O(N^2 \log(N))$ . Finally, since there is at most  $N$  durations, we need at most  $O(N)$  space to store the schedule of each transactions.

### 4.3 The GenerateLoadNoC procedure

As mentioned in Section 4.1, the second condition for POGen to work is that: procedure GenerateLoadNoC can generate a feasible load set  $\{l_i^k : \forall \tau_i \in \mathcal{T}\}$  for interval  $\text{int}^k$  when its inputs satisfy Inequalities 4.3 and 4.4. A feasible load set is one that satisfies Inequalities 4.1 and 4.2. In this section, we will present a version of GenerateLoadNoC. There are two questions that have to be answered: (1) is there a feasible load set? (2) is there an efficient algorithm to find it? We will show that the problem of finding a feasible load set is equivalent to the problem *Circulations in Graphs with Demands and Lower bounds* [14] where the demand at every vertex is 0. This is the problem of finding a feasible circulation flow in a directed graph  $G$  whose each edge has a capacity and a lower bound. Furthermore, we will prove that if the utilization of each PO-set is smaller than the utilization bound expressed by Inequalities 4.5, there always exists a feasible solution therefore answering Question 1. Then, since the Ford-Fulkerson algorithm [14] can be used to solve the problem, Question 2 is also answered.

In the following, we will describe the construction of the directed graph  $G$  from the input of GenerateLoadNoC. Graph  $G$  is constructed such that: 1) the value of the flow on an edge represents the interval load of a transaction or the total interval load of a PO-set. Flow values are subject to the same upper bounds and lower bounds of the loads they represent (as in Inequalities 4.1 and 4.2); 2) the equality between the sum of the input-flow values and the sum of the output-flow values at a vertex represents the equality between the total interval load of some PO-sets and the sum of the interval loads of the transactions in these PO-sets. The construction guarantees that the value of a feasible flow on an edge is also a feasible interval load of the transaction or the PO-set it represents.

**Graph construction:** we define a tuple  $G = (V, E)$  as follows:

Vertices of  $G$ :

- For each PO-set  $\mathcal{D}_l$ , define a vertex  $v_l$ .
- The set of vertices of  $G$  is  $V = \{v^*, v_l : l \in [1, N^{\mathcal{D}}]\}$  where  $v^*$  is an additional vertex.

Edges and flow values of  $G$ :

- For each PO-set  $\mathcal{D}_l$ , define a directed edge  $g_l^{\mathcal{D}}$  which is called a *PO-set edge*. Furthermore, define for each edge  $g_l^{\mathcal{D}}$  two integer constants  $c_l^{\mathcal{D}}$  and  $b_l^{\mathcal{D}}$  which are called the capacity and the lower bound of edge  $g_l^{\mathcal{D}}$  where  $c_l^{\mathcal{D}} = \min(|\text{int}^k|, \lceil \text{lag}(\mathcal{D}_l, \text{int}^k) \rceil)$  and  $b_l^{\mathcal{D}} = \lfloor \text{lag}(\mathcal{D}_l, \text{int}^k) \rfloor$ . Finally, define for each edge  $g_l^{\mathcal{D}}$  a real variable  $x_l^{\mathcal{D}}$  which is called the flow value of the edge. The flow value is subject to the constraints in Inequality 4.6 (which is the same as Inequality 4.2):

$$\forall \mathcal{D}_l \subset \mathcal{T} : b_l^{\mathcal{D}} \leq x_l^{\mathcal{D}} \leq c_l^{\mathcal{D}}. \quad (4.6)$$

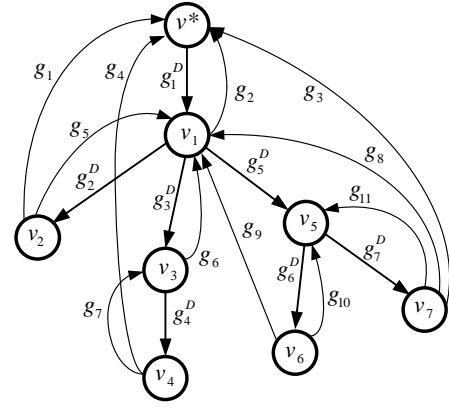
- For each transaction  $\tau_i$ , define a directed edge  $g_i$  which is called a *transaction edge*. Furthermore, define for each edge  $g_i$  two integer constants  $c_i$  and  $b_i$  which are called the capacity and the lower bound of edge  $g_i$  where  $c_i = \lceil \text{lag}(\tau_i, \text{int}^k) \rceil$  and  $b_i = \lfloor \text{lag}(\tau_i, \text{int}^k) \rfloor$ . Finally, define for each edge  $g_i$  a real variable  $x_i$  which is called the flow value of the edge. The flow value is subject to the constraints in Inequality 4.7 (which is the same as Inequality 4.1):

$$\forall \tau_i \in \mathcal{T} : b_i \leq x_i \leq c_i. \quad (4.7)$$

- The set of edges of  $G$  is:  $E = \{g_i : i \in [1, N]\} \cup \{g_l^{\mathcal{D}} : j \in [1, N^{\mathcal{D}}]\}$ . The total number of edges is  $|E| = N + N^{\mathcal{D}}$ .

Rules for directing edges:

- The set of edges that enter  $v^*$  is  $\text{Enter}^* = \{g_i : \tau_i \in \mathcal{D}_1\}$ ; the set of edges that exit  $v^*$  is  $g_1^{\mathcal{D}}$ .
- The set of edges that enter  $v_l$  is PO-set edge  $g_l^{\mathcal{D}}$  and transaction edges representing transactions that are in children of  $\mathcal{D}_l$  but not in  $\mathcal{D}_l$ , i.e.  $\text{Enter}_l = \bigcup_{\mathcal{D}_m \in \text{children}(\mathcal{D}_l)} \{g_i : \tau_i \in \mathcal{D}_m \setminus \mathcal{D}_l\}$ .
- The set of edges that exit  $v_l$  are PO-set edges  $\{g_m^{\mathcal{D}} : \mathcal{D}_m \in \text{children}(\mathcal{D}_l)\}$  and transaction edges representing transactions that are in  $\mathcal{D}_l$  but not in children of  $\mathcal{D}_l$ , i.e.  $\text{Exit}_l = \{g_i : \tau_i \in \mathcal{D}_l \setminus \bigcup_{\mathcal{D}_m \in \text{children}(\mathcal{D}_l)} \mathcal{D}_m\}$ .



**Figure 5. Graph  $G$  of  $\mathcal{T}$  shown in Figure 1**

Flow conservation constraint:

- The flow values in graph  $G$  is subject to the flow conservation constraint [14] in which given a vertex, the sum of the flow values entering it minus the sum of the flow values exiting it is zero.

Figure 5 shows graph  $G$  constructed from the transaction set shown in Figure 1. Consider vertex  $v_1$ . Edges that enter  $v_1$  are  $g_1^{\mathcal{D}}$  and  $\text{Enter}_1 = \{g_5, g_6, g_8, g_9\}$ . Edges in  $\text{Enter}_1$  are edges that represent transactions which belong to  $\text{children}(\mathcal{D}_1) = \{\mathcal{D}_2, \mathcal{D}_3, \mathcal{D}_5\}$  but do not belong to  $\mathcal{D}_1$ . Furthermore, edges that exit  $v_1$  are  $\{g_2^{\mathcal{D}}, g_3^{\mathcal{D}}, g_5^{\mathcal{D}}\}$  and  $\text{Exit}_1 = \{g_2\}$ . As we will prove later, this construction guarantees that the equality between the sum of the input-flow values and the sum of the output-flow values at  $v_1$  (due to the flow conservation constraint) represents the equality between the sum of interval loads of transactions in every child of  $\mathcal{D}_1$  and the total interval load of its children.

We will prove in Lemma 4.6 that graph  $G$  is indeed a directed graph in which every edge is directed and has two endpoints. The proof will use Lemma 4.4, and 4.5.

**Lemma 4.4** For every  $\mathcal{D}_l, \mathcal{D}_m : \mathcal{D}_l \neq \mathcal{D}_m$ , the following is true:

$$(\mathcal{D}_l \setminus \text{parent}(\mathcal{D}_l)) \cap (\mathcal{D}_m \setminus \text{parent}(\mathcal{D}_m)) = \emptyset. \quad (4.8)$$

**Proof.**

Since the lemma is trivial when  $\mathcal{D}_l = \text{parent}(\mathcal{D}_m)$  or  $\mathcal{D}_m = \text{parent}(\mathcal{D}_l)$ , we assume that these conditions are false. If  $\mathcal{D}_l \not\prec \mathcal{D}_m$  and  $\mathcal{D}_m \not\prec \mathcal{D}_l$ , then the lemma is true because of Property 3.2. If  $\mathcal{D}_l \prec \mathcal{D}_m$ , since  $\mathcal{D}_l \neq \text{parent}(\mathcal{D}_m)$ , we have  $\mathcal{D}_l \prec \text{parent}(\mathcal{D}_m) \prec \mathcal{D}_m$ . By Property 3.1, for every  $\tau_i$  where  $\tau_i \in \mathcal{D}_l$  and  $\tau_i \in \mathcal{D}_m$  we have  $\tau_i \in \text{parent}(\mathcal{D}_m)$ . Therefore  $\mathcal{D}_l \cap (\mathcal{D}_m \setminus \text{parent}(\mathcal{D}_m)) = \emptyset$ , which proves the lemma. Finally, using similar technique and interchanging  $\mathcal{D}_l$  with  $\mathcal{D}_m$ , we can also prove the lemma when  $\mathcal{D}_m \prec \mathcal{D}_l$ .  $\square$

**Lemma 4.5** For every  $\mathcal{D}_l, \mathcal{D}_m : \mathcal{D}_l \neq \mathcal{D}_m$ , the following is true:

$$\left\{ \mathcal{D}_l \setminus \bigcup_{\mathcal{D}_n \in \text{children}(\mathcal{D}_l)} \mathcal{D}_n \right\} \cap \left\{ \mathcal{D}_m \setminus \bigcup_{\mathcal{D}_n \in \text{children}(\mathcal{D}_m)} \mathcal{D}_n \right\} = \emptyset. \quad (4.9)$$

**Proof.**

Since the lemma is trivial when  $\mathcal{D}_l = \text{parent}(\mathcal{D}_m)$  or  $\mathcal{D}_m =$

parent( $\mathcal{D}_l$ ), we assume that these conditions are false. If  $\mathcal{D}_l \not\prec \mathcal{D}_m$  and  $\mathcal{D}_m \not\prec \mathcal{D}_l$ , then the lemma is true because of Property 3.2. If  $\mathcal{D}_l \prec \mathcal{D}_m$ , since  $\mathcal{D}_l \neq \text{parent}(\mathcal{D}_m)$ , we have there exists  $\mathcal{D}_n \in \text{children}(\mathcal{D}_l)$  where  $\mathcal{D}_l \prec \mathcal{D}_n \prec \mathcal{D}_m$ . By Property 3.1, for every  $\tau_i$  where  $\tau_i \in \mathcal{D}_l$  and  $\tau_i \in \mathcal{D}_m$  we have  $\tau_i \in \mathcal{D}_n$ . Therefore  $(\mathcal{D}_l \setminus \mathcal{D}_n) \cap \mathcal{D}_m = \emptyset$ , which proves the lemma. Finally, using similar technique and interchanging  $\mathcal{D}_l$  with  $\mathcal{D}_m$ , we can also prove the lemma when  $\mathcal{D}_m \prec \mathcal{D}_l$ .  $\square$

**Lemma 4.6**  $G$  is a directed graph where  $|E| \leq 2N$ .

**Proof.**

Since every edge of  $G$  is directed, it remains to show that each edge enters one and only one vertex and exits one and only one vertex. Note that there is one edge defined for each PO-set and one edge defined for each transaction.

Consider PO-set edges: By the rules of directing edges, PO-set edge  $g_l^D$  exits only vertex  $v^*$ . Furthermore, if  $\mathcal{D}_l = \text{parent}(\mathcal{D}_m)$ , the PO-set edge  $g_m^D$  exits vertex  $v_l$ . Since each  $\mathcal{D}_m$  where  $\mathcal{D}_m \neq \mathcal{D}_l$  has one and only one parent, the PO-set edge  $g_m^D$  exits one and only one vertex. Therefore, each PO-set edge exits one and only one vertex. In addition, each PO-set edge  $g_l^D$  enters only vertex  $v_l$ . Therefore each PO-set edge enters one and only one vertex.

Consider transaction edges: By the rules of directing edges, the set of transaction edges that enter  $v^*$  is  $\text{Enter}^*$  and enter  $v_l$  is  $\text{Enter}_l$ . By Lemma 4.4, the collection of sets  $\{\text{Enter}^*, \text{Enter}_l : \forall l\}$  is pairwise disjoint. Therefore each transaction edge  $g_i$  will appear in only one of the sets in the collection. Furthermore, since  $\bigcup_{\mathcal{D}_m \in \mathcal{T}} \mathcal{D}_m \setminus \text{parent}(\mathcal{D}_m) = \mathcal{T}$ , each transaction edge  $g_i$  must appear in at least one of the sets in the collection  $\{\text{Enter}^*, \text{Enter}_l : \forall l\}$ . In other words, each transaction edge  $g_i$  enters one and only one vertex.

By the rules of directing edges, the set of transaction edges that exits  $v^*$  is empty and exits  $v_l$  is  $\text{Exit}_l$ . By Lemma 4.5, the collection of sets  $\{\text{Exit}_l : \forall l\}$  is pairwise disjoint. Therefore each transaction edge  $g_i$  will appear in only one of the sets in the collection. Furthermore, since  $\bigcup_{\mathcal{D}_l \in \mathcal{T}} (\mathcal{D}_l \setminus \bigcup_{\mathcal{D}_m \in \text{children}(\mathcal{D}_l)} \mathcal{D}_m) = \mathcal{T}$ , each transaction edge  $g_i$  must appear in at least one of the sets in the collection  $\{\text{Exit}_l : \forall l\}$ . In other words, each transaction edge  $g_i$  exits one and only one vertex.

Now we will prove that  $|E| \leq 2N$ , since each transaction edge exits one and only one vertex and each vertex except  $v^*$  has at least one transaction edge exists from it, we have  $|V \setminus \{v^*\}| \leq N$ . Since by definition  $|V \setminus \{v^*\}| = N^D$  and  $|E| = N + N^D$ , we have  $|E| \leq 2N$   $\square$

It remains to show that `GenerateLoadNoC` honors Condition 4.2. For simplicity of exposition, we split the proof in multiple lemmas. First, Lemma 4.7 proves an important property of graph  $G$  regarding the flow values. Then, this property will be used to prove in Lemma 4.8 that graph  $G$  has a feasible flow if Inequalities 4.3, 4.4 are satisfied for interval  $\text{int}^k$  and furthermore all PO-sets satisfy the utilization bound shown in Inequality 4.5. Note that we know from [14] that if graph  $G$  has a feasible flow, then it has an integral feasible flow which can be found by the Ford-Fulkerson algorithm [14]. Therefore, to complete the proof, we will have to prove that a feasible load set can be derived from an integral feasible flow of  $G$  (Lemma 4.9).

**Lemma 4.7** A flow in graph  $G$  honors the flow conservation constraint at every vertex  $v_l$  if and only if the following equalities hold

for every PO-set  $\mathcal{D}_l$ :

$$\sum_{\tau_i \in \mathcal{D}_l} x_i = x_l^D. \quad (4.10)$$

**Proof.**

We prove the lemma by induction.

Basis case: Consider  $v_l$  where  $\text{height}(\mathcal{D}_l) = 0$  or equivalently  $\text{children}(\mathcal{D}_l) = \emptyset$ . By the rules of directing edges, we have the set of edges that enter  $v_l$  is  $\{g_l^D\}$  and the set of edges that exit  $v_l$  is  $\{g_i : \tau_i \in \mathcal{D}_l\}$ . Therefore, the flow conservation constraint holds at vertex  $v_l$  if and only if  $\sum_{\tau_i \in \mathcal{D}_l} x_i = x_l^D$ .

Induction case: Assume that the hypothesis is true  $\forall \mathcal{D}_l$  where  $\text{height}(\mathcal{D}_l) \leq H - 1$ . We will prove that the hypothesis is also true  $\forall \mathcal{D}_l$  where  $\text{height}(\mathcal{D}_l) = H$ . By definition of the tree node height, the induction hypothesis implies that  $\forall \mathcal{D}_m \in \text{children}(\mathcal{D}_l)$ ,  $v_m$  honors the flow conservation constraint if and only if:

$$\sum_{\tau_i \in \mathcal{D}_m} x_i = x_m^D. \quad (4.11)$$

By the rules of directing edges and Lemma 4.4, the total value of flows that enters vertex  $v_l$  is:

$$x_j^{\text{enter}} = \sum_{\mathcal{D}_m \in \text{children}(\mathcal{D}_l)} \sum_{\tau_i \in \mathcal{D}_m \setminus \mathcal{D}_l} x_i + x_l^D,$$

and the total value of flows that exits vertex  $v_l$  is:

$$x_j^{\text{exit}} = \sum_{\tau_i \in \mathcal{D}_l \setminus \bigcup_{\mathcal{D}_m \in \text{children}(\mathcal{D}_l)} \mathcal{D}_m} x_i + \sum_{\mathcal{D}_m \in \text{children}(\mathcal{D}_l)} x_m^D.$$

To complete the induction step, it remains to show that  $x_j^{\text{enter}} = x_j^{\text{exit}}$  if and only if Equation 4.10 holds. Note that by Equation 4.11,  $x_j^{\text{enter}} = x_j^{\text{exit}}$  is equivalent to:

$$\begin{aligned} x_l^D &= - \sum_{\mathcal{D}_m \in \text{children}(\mathcal{D}_l)} \sum_{\tau_i \in \mathcal{D}_m \setminus \mathcal{D}_l} x_i \\ &+ \sum_{\tau_i \in \mathcal{D}_l \setminus \bigcup_{\mathcal{D}_m \in \text{children}(\mathcal{D}_l)} \mathcal{D}_m} x_i + \sum_{\mathcal{D}_m \in \text{children}(\mathcal{D}_l)} \sum_{\tau_i \in \mathcal{D}_m} x_i \\ &= \sum_{\mathcal{D}_m \in \text{children}(\mathcal{D}_l)} \sum_{\tau_i \in \mathcal{D}_m \cap \mathcal{D}_l} x_i + \sum_{\tau_i \in \mathcal{D}_l \setminus \bigcup_{\mathcal{D}_m \in \text{children}(\mathcal{D}_l)} \mathcal{D}_m} x_i \end{aligned} \quad (4.12)$$

Finally, since by Property 3.2, PO-sets in  $\text{children}(\mathcal{D}_l)$  are pairwise disjointed sets, Equation 4.12 is equivalent to:

$$\begin{aligned} x_l^D &= \sum_{\tau_i \in \mathcal{D}_l \cap \left( \bigcup_{\mathcal{D}_m \in \text{children}(\mathcal{D}_l)} \mathcal{D}_m \right)} x_i \\ &+ \sum_{\tau_i \in \mathcal{D}_l \setminus \bigcup_{\mathcal{D}_m \in \text{children}(\mathcal{D}_l)} \mathcal{D}_m} x_i = \sum_{\tau_i \in \mathcal{D}_l} x_i \end{aligned}$$

This completes the proof.  $\square$

**Lemma 4.8** There exists a feasible flow in graph  $G$  if Inequalities 4.3, 4.4 are satisfied for interval  $\text{int}^k$  and all PO-sets satisfy the utilization bound in Inequality 4.5.

**Proof.**

First note that Inequalities 4.3 are necessary for the edge constraints

on each PO-set edge (Inequality 4.6) to be satisfied. Let us construct a flow as follows.

$$\begin{aligned} \forall \tau_i \in \mathcal{T} : x_i &= \text{lag}(\tau_i, \text{int}^k) \\ \forall \mathcal{D}_l \subset \mathcal{T} : x_l^{\mathcal{D}} &= \text{lag}(\mathcal{D}_l, \text{int}^k) \end{aligned}$$

We will have to prove that the constructed flow satisfies the edge constraints and the flow conservation constraints. Given the constructed flow, it is easy to verify that the edge constraints of each transaction edge (Inequality 4.7) and the left-side edge constraints of each PO-set edge (Inequality 4.6) are satisfied. The right-side edge constraints of each PO-set edge are satisfied because by the definition of the lag function and by Inequalities 4.4, before the execution of GenerateLoadNoC for interval  $\text{int}^k$  we have the following:

$$\begin{aligned} \text{lag}(\mathcal{D}_l, \text{int}^k) &= u_l^{\mathcal{D}} * t^{k+1} - \sum_{\tau_i \in \mathcal{D}_l} \sum_{x \in [0, t^k]} S(\tau_i, x) \\ &\leq u_l^{\mathcal{D}} * t^{k+1} - \lfloor u_l^{\mathcal{D}} * t^k \rfloor \\ &< u_l^{\mathcal{D}} * t^{k+1} - u_l^{\mathcal{D}} * t^k + 1. \end{aligned} \quad (4.13)$$

Since  $L \leq \min_k(\text{int}^k)$ , we have  $\frac{L-1}{L} \leq \frac{|\text{int}^k| - 1}{|\text{int}^k|}$ . Hence, by

Inequality 4.5, we have  $u_l^{\mathcal{D}} \leq \frac{|\text{int}^k| - 1}{|\text{int}^k|}$ . Apply this inequality to Inequality 4.13, we have:

$$\text{lag}(\mathcal{D}_l, \text{int}^k) < u_l^{\mathcal{D}} * t^{k+1} - u_l^{\mathcal{D}} * t^k + 1 \leq |\text{int}^k|.$$

Since furthermore  $\text{lag}(\mathcal{D}_l, \text{int}^k) \leq \lceil \text{lag}(\mathcal{D}_l, \text{int}^k) \rceil$ , it follows that  $\text{lag}(\mathcal{D}_l, \text{int}^k) \leq \min(|\text{int}^k|, \lceil \text{lag}(\mathcal{D}_l, \text{int}^k) \rceil)$ .

It remains to verify that the flow conservation constraint is honored at every vertex. Since the constructed flow satisfied Equation 4.10, the sufficient condition of Lemma 4.7 proves this statement.  $\square$

**Lemma 4.9** *If there is an integral feasible flow in graph  $G$ , then there is a feasible load set where*

$$\forall \tau_i \in \mathcal{T} : l_i^k = x_i \quad (4.14)$$

**Proof.**

We have to prove that the load set where  $\forall \tau_i \in \mathcal{T} : l_i^k = x_i$  satisfies Inequalities 4.1 and 4.2. By the transaction edge constraints in Inequality 4.7, the following inequality holds.

$$\forall \tau_i \in \mathcal{T} : \lfloor \text{lag}(\tau_i, \text{int}^k) \rfloor \leq l_i^k \leq \lceil \text{lag}(\tau_i, \text{int}^k) \rceil$$

Thus the interval loads satisfy Inequality 4.1.

Furthermore, by the necessary condition of Lemma 4.7, we have:

$$\forall \mathcal{D}_l \in \mathcal{T} : \sum_{\tau_i \in \mathcal{D}_l} x_i = x_l^{\mathcal{D}}. \quad (4.15)$$

Then since  $\sum_{\tau_i \in \mathcal{D}_l} l_i^k = \sum_{\tau_i \in \mathcal{D}_l} x_i$  and  $x_l^{\mathcal{D}}$  is subject to PO-set edge constraints in Inequality 4.6, Inequality 4.2 is satisfied.  $\square$

**Algorithm analysis:** Since  $\forall g_i \in E : c_i - b_i \leq 1$ ,  $\forall g_l^{\mathcal{D}} \in E : c_l^{\mathcal{D}} - b_l^{\mathcal{D}} \leq 1$  and  $N^{\mathcal{D}} \leq N$ , we have  $\Delta = \sum_{g_i \in E} (c_i - b_i) + \sum_{g_l^{\mathcal{D}} \in E} (c_l^{\mathcal{D}} - b_l^{\mathcal{D}}) \leq 2N$ . The time complexity of the Ford-Fulkerson algorithm in finding a

feasible circulation in graph  $G$  is  $O(|E|f^{max})$  where  $f^{max}$  is the maximum flow value of a graph derived from  $G$  and  $f^{max} \leq \Delta$  (see [14] for details). Since  $\Delta \leq 2N$  and  $|E| \leq 2N$ , the time complexity of GenerateLoadNoC is  $O(N^2)$ . Finally, since the time complexity of POBaseNoC is  $O(N^2 \log(N))$ , the time complexity of POGen to generate the schedule for each scheduling interval is  $O(N^2 \log(N))$ .

**The sufficient utilization bound of POGen:** As shown in Lemma 4.8, GenerateLoadNoC induces a sufficient utilization bound shown in Inequality 4.5. Since this bound is stricter than that of POBaseNoC, it becomes a sufficient utilization bound of POGen.

## 5 Implementation

In this section, we discuss some practical factors in the implementation of POGen in real systems and show the experimental measurement of the algorithm execution overheads.

An important parameter in the operation of POGen is the size of  $L$  in number of slots because it dictates the POGen's sufficient utilization bound (shown in Inequalities 4.5). Note that the real-time value of  $L$  is determined by the applications. As shown in [18], typical real-time applications have periods that are multiple of milliseconds. Therefore, we believe it can be assumed that the greatest common divisor of all transaction periods is at least 1ms. Meanwhile a typical modern SoC bus [2, 11, 12] has bus clock frequency no smaller than 100MHz. Therefore, it is reasonable to select the slot size to be no bigger than 10us (equivalent to 1000 bus cycles), which results in the value of  $L$  in number of slots to be no smaller than 100. The utilization bound, therefore, will be 0.99. It is worth noting that the size of a slot does *not* affect the algorithm overhead because the time complexity POGen only depends on the number of transactions  $N$ .

Implementing our scheduling framework requires two components: 1) POGen is executed on each processing element at the beginning of each interval to generate the interval's schedule and 2) a *slot scheduler* transmits a given transaction in its assigned slots according to the generated interval schedule. The implementation of the slot scheduler depends on the NoC architecture. In a software controlled NoC such as in the CellBE processor [2], the slot scheduler can be implemented in software on each processing element by using an interrupt handler to trigger the scheduler at the beginning of each duration; since there are at most  $N$  durations, on each processing element there are at most  $N$  interrupts within each scheduling intervals. In a custom NoC, the slot scheduler could be implemented in the router connected to the processing element.

In order to measure POGen overhead in the real SoC systems, we implemented POGen on a IBM CellBE processor platform [10]. We selected Cell processor because it represents a typical modern SoC whose components are interconnected by a software controllable NoC. We generated random transaction sets using the same method described in Section 6. Table 1 shows the average and maximum execution time of POGen given various transaction sets size  $N$ . Given the smallest scheduling interval to be 1ms, the overhead is less than 1.5% of the scheduling interval size.

$N$	10	20	30	40
Average (us)	4.5	6.4	9.5	12.1
Max (us)	6.9	7.7	11.2	13.8

**Table 1.** POGen execution overhead

## 6 Evaluation

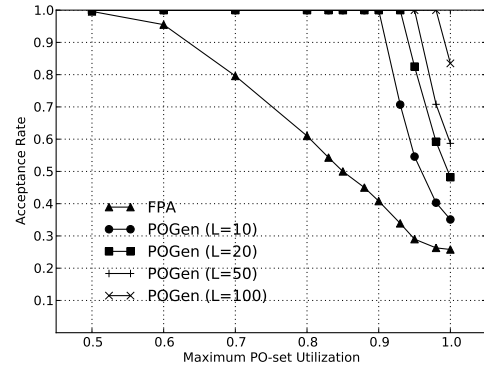
Most of the previous related works [20, 21, 3, 16, 24] have focused on the Fixed-priority Scheduling Algorithm (FPA). These works deal with the methods for schedulability analysis and priority assignment. More specifically, Shi et al. have recently proposed in [20] a branch-and-bound algorithm that searches for a feasible priority set for a transaction set. If a feasible priority set exists then the transaction set is guaranteed to be schedulable under worst-case transaction latency (WTL) analysis proposed in [21] the best of our knowledge, the works in [21, 20] are the state of art.

In this section, we are interested in comparing the performance of POGen with the solution proposed in [21, 20] assuming acyclic transaction sets<sup>2</sup>. The concerned performance metric is the acceptance rate of POGen and the FPA where the acceptance rate is calculated as the number of schedulable transaction sets over the number of generated transaction sets. A transaction set is schedulable under POGen if it passes the utilization bound test show Inequality 4.5, whereas it is schedulable under the FPA if it has a feasible priority set generated by the algorithm in [20].

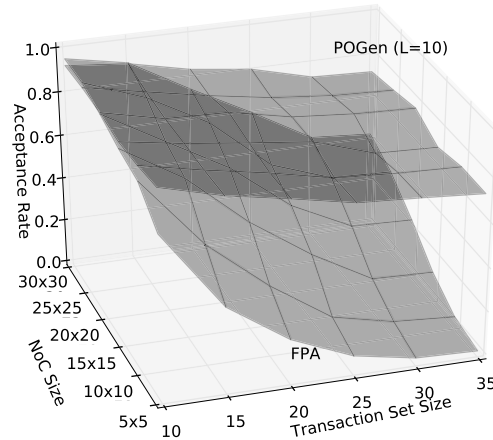
To generate random transaction sets for the experiments, we used similar parameters and methods used in [20] except that we are only concerned with acyclic transaction sets. The transactions' sources and destinations are randomly generated on a 2D mesh and transactions are routed using the dimension-ordered Y routing. Like [20], we use the maximum PO-set utilization of a transaction set (which is "the maximum link utilization" in [20]) as a controlled variable. Given a maximum PO-set utilization  $u^{max}$ , the utilization of transactions is generated according to the uniform distribution algorithm in [5] such that the utilizations of all PO-sets are no larger than  $u^{max}$ . The transmission time  $e_i$  of transaction  $\tau_i$  is a uniformly-distributed random number in the range from 1 to 1024 slots. The period  $p_i$  is then determined as a multiple of  $L$  using the following formula:  $\lceil e_i / (u_i * L) \rceil * L$ . Given a pair of  $\{e_i, p_i\}$ , we recalculate  $u_i$  to be  $e_i / p_i$ . Following the discussion in Section 5, we generated transaction sets with  $L$  to be 10, 20, 50, and 100 slots. In the following experiments, there are 1000 transaction sets generated at each measurement point.

Figure 6 shows the acceptance rates of FPA and POGen with various maximum PO-set utilization. The size of NoC is 10x10 and each transaction set has 20 transactions. For POGen, we report the results with  $L$  equals to 10, 20, 50, 100. The better performance of POGen comes from the fact that the WTL analysis in [21] does not take advantage of the parallelism between non-overlapping transactions. For example, consider the transaction set shown in Figure 1. Assume  $\tau_6$  and  $\tau_7$  have higher priority than  $\tau_4$ . According to the WTL analysis in [21], the interference of transactions  $\tau_6$  and  $\tau_7$  on the execution of  $\tau_4$  is calculated as if all transactions were using a single-shared resource. However, POGen allows  $\tau_6$  and  $\tau_7$  to be executed in parallel as shown in Figure 4.

Figure 7 shows the acceptance rate of FPA and POGen with various transaction set sizes and NoC sizes. We report the result where



**Figure 6.** Acceptance rate versus PO-set utilizations



**Figure 7.** Acceptance rate versus NoC sizes and  $N$

$L$  equals to 10 slots and the maximum PO-set utilization is 0.95. In most cases, the acceptance rate of POGen is higher than that of FPA especially when transaction set size is higher and the NoC size is smaller. The reason is that in these situations, there are more transaction overlaps. Therefore, FPA suffers more from the effect described in the previous paragraph.

## References

- [1] Cell be programming tutorial. IBM, 2007.
- [2] T. W. Ainsworth and T. M. Pinkston. Characterizing the cell eib on-chip network. *IEEE Micro*, 27(5):6–14, 2007.
- [3] S. Balakrishnan and F. Özgüner. A priority-driven flow control mechanism for real-time traffic in multiprocessor networks. *IEEE Trans. Parallel Distrib. Syst.*, 9(7):664–678, 1998.
- [4] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. A. Varvel. Proportionate progress: a notion of fairness in resource allocation. In *STOC '93: Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 345–354, New York, NY, USA, 1993. ACM.
- [5] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Syst.*, 30(1-2), 2005.
- [6] T. Bjerregaard and S. Mahadevan. A survey of research and practices of network-on-chip. *ACM Comput. Surv.*, 38(1):1, 2006.
- [7] B. D. Bui, R. Pellizzoni, and M. Caccamo. Real-time scheduling of concurrent transactions in multi-domain ring buses. Technical report, University of Illinois at Urbana Champaign, 2010. <http://hdl.handle.net/2142/17451>.

<sup>2</sup>Note that algorithms in [21, 20] can also be used for cyclic transaction sets.

- [8] B. D. Bui, R. Pellizzoni, and M. Caccamo. Real-time scheduling of concurrent transactions in NoC (<http://www.cs.uiuc.edu/homes/bachbui2/publications/nocsched-full.pdf>). Technical report, University of Illinois at Urbana Champaign, 2010.
- [9] B. D. Bui, R. Pellizzoni, D. K. Chivukula, and M. Caccamo. Real-time scheduling on multi-domain ring buses. In *RTCSA '10: Proceedings of the 14th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Macau, China, 2010.
- [10] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: A performance view. IBM Research, 2005.
- [11] J. Howard et al. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *IEEE International Solid-State Circuits Conference*, 2010.
- [12] K. Goossens, J. Dielissen, and A. Radulescu. Aethereal network on chip: concepts, architectures, and implementations. *Design Test of Computers, IEEE*, 22(5):414 – 421, 2005.
- [13] S. Gopalakrishnan, L. Sha, and M. Caccamo. Hard real-time communication in bus-based networks. In *RTSS '04: Proceedings of the 25th IEEE International Real-Time Systems Symposium*, pages 405–414, Washington, DC, USA, 2004. IEEE Computer Society.
- [14] J. Kleinberg and É. Tardos. *Algorithm Design*. Addison Wesley, March 2005.
- [15] J. P. Lehoczky and L. Sha. Performance of real-time bus scheduling algorithms. *SIGMETRICS Perform. Eval. Rev.*, 14(1):44–53, 1986.
- [16] J. P. Li and M. W. Mutka. Real-time virtual channel flow control. *J. Parallel Distrib. Comput.*, 32(1):49–65, 1996.
- [17] C. L. Liu and W. J. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.
- [18] C. D. Locke, D. R. Vogel, L. Lucas, and J. B. Goodenough. Generic avionics software specification. Technical Report CMU/SEI-90-TR-8, 1990.
- [19] M. D. Natale and A. Meschi. Scheduling messages with earliest deadline techniques. *Real-Time Systems*, 20(3):255–285, 2001.
- [20] Z. Shi and A. Burns. Priority assignment for real-time wormhole communication in on-chip networks. In *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium*, Washington, DC, USA, 2008.
- [21] Z. Shi and A. Burns. Real-time communication analysis for on-chip networks with wormhole switching. In *NOCS '08: Proceedings of the Second ACM/IEEE International Symposium on Networks-on-Chip*, pages 161–170, Washington, DC, USA, 2008. IEEE Computer Society.
- [22] K. Tindell, A. Burns, and A. Wellings. Analysis of hard real-time communications. *Real-Time Systems*, 9:147–171, 1995.
- [23] D. Wiklund and D. Liu. Design mapping, and simulations of a 3g wcdma/fdd basestation using network on chip. *System-on-Chip for Real-Time Applications, International Workshop on*, 0:252–256, 2005.
- [24] L. Zhonghai, J. Axel, and S. Ingo. Feasibility analysis of messages for on-chip networks using wormhole routing. In *ASP-DAC '05: Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, Shanghai, China, 2005.
- [25] D. Zhu, D. Mosse, and R. Melhem. Multiple-resource periodic scheduling problem: How much fairness is necessary. In *24th IEEE International Real-Time Systems Symposium*, 2003.

## 7 Appendix

**Proof that  $N^{\mathcal{D}} \leq N$ :** By the defined PO-set index scheme and by Property 3.1 and 3.2, we have transactions in  $\mathcal{D}_j \setminus \text{parent}(\mathcal{D}_j)$  do not belong to  $\bigcup_{i \in [1, j-1]} \mathcal{D}_i$ . Therefore sets in collection  $A = \{\mathcal{D}_j \setminus \text{parent}(\mathcal{D}_j) : j \in [1, N^{\mathcal{D}}]\}$  are mutually disjoint. Since the size of  $\mathcal{T}$  is  $N$ , by pigeonhole principle, we have  $|A| = N^{\mathcal{D}} \leq N$ .

**Proof Property 3.3:** assume by contradiction that  $\mathcal{D}_l \not\leq \mathcal{D}_m$ . If  $\mathcal{D}_m < \mathcal{D}_l$ , then  $m < l$ . If otherwise  $\mathcal{D}_m \not\leq \mathcal{D}_l$ , then either  $m < l$  or  $m > n$ . Both cases contradict the property's assumption.

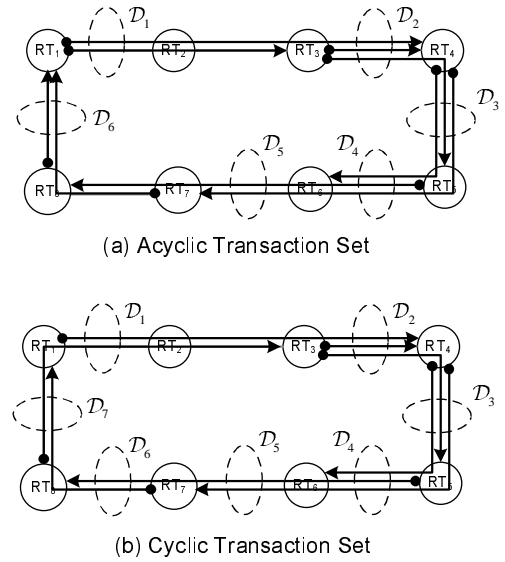


Figure 8. A transaction set in NoC

**Proof Property 3.4:** assume by contradiction that there exists  $\mathcal{D}_l$  where  $\tau_i \in \mathcal{D}_l$  and  $\min\text{PO}_i \not\leq \mathcal{D}_l$ . Since, by definition,  $\min\text{id}_i < l$ , we have  $\mathcal{D}_l \not\leq \min\text{PO}_i$ . Then, by Property 3.2,  $\mathcal{D}_l$  cannot share  $\tau_i$  with  $\min\text{PO}_i$ , which contradicts the property's assumption.

**Lemma 7.1** A transaction set that is cyclic or acyclic on NoC with ring topology by the definition in [9, 7] is also cyclic or acyclic, respectively, by the definition proposed in Section 3.

### Proof.

By definition in [9, 7], a transaction set in NoC with ring topology is acyclic if there exists a router (called *start router*) on the ring where there is no transaction going through and is cyclic otherwise. Figure 8(a) shows an example of an acyclic transaction set by this definition where  $RT_1$  does not have any transaction going through. Figure 8(b) shows an example of a cyclic transaction set. Note that, by the definition of a PO-set, for each PO-set on the NoC with ring-topology, there is a set of contiguous physical links which are used by all transactions in the PO-set. Let call these links of  $\mathcal{D}_i$ , the *common set* of  $\mathcal{D}_i$ . For example, the common physical links of  $\mathcal{D}_1$  in Figure 8(a) are links  $RT_1 \rightarrow RT_2$  and  $RT_2 \rightarrow RT_3$ . Also by definition, two PO-sets do not share a common links because otherwise the union of these two PO-sets is also a PO-set thus the two original PO-sets are not maximal sets.

Consider a ring-topology NoC, let index its routers and physical links increasingly in clockwise direction and if the transaction set is acyclic then the start router has the smallest index. Then, let index the PO-sets such that a PO-set has smaller index if its common physical links have smaller indexes. Indexes shown in Figure 8(a) and (b) conform to this index scheme. Note that this PO-set index scheme definition is different with the definition in Section 3. We introduce this for ease of presentation.

We will now prove that if a transaction set on ring-topology NoC is acyclic by definition in [9, 7], then we can construct a PO-tree of this transaction set that satisfies Property 3.1 and 3.2. Given the defined index scheme, consider an acyclic transaction set  $\mathcal{T}$  by definition in [9, 7] and assume that its PO-graph is connected. Consider

the PO-tree of  $\mathcal{T}$  which is rooted at  $\mathcal{D}_1$  and is built from its PO-graph by removing all edges in the PO-graph except for the edge between  $\mathcal{D}_l$  and  $\mathcal{D}_{l+1}$  where  $i \in [1, N^{\mathcal{D}}]$ . Note that in this PO-tree, each PO-set  $\mathcal{D}_l$  has only one child which is  $\mathcal{D}_{l+1}$ , therefore it cannot violate Property 3.2 because there are no two PO-sets  $\mathcal{D}_l$  and  $\mathcal{D}_m$  where  $\mathcal{D}_l \not\prec \mathcal{D}_m$  and  $\mathcal{D}_m \not\prec \mathcal{D}_l$ . We prove this PO-tree also satisfies Property 3.1 as follows. Consider  $\mathcal{D}_l \prec \mathcal{D}_m \prec \mathcal{D}_n$ . By the way the PO-tree is built, we have  $l < m < n$ . Then by the defined PO-set index scheme on ring-topology NoC, we have for every  $\tau_i$  if  $\tau_i \in \mathcal{D}_l$  and  $\tau_i \in \mathcal{D}_n$ , and  $l < m < n$ , then  $\tau_i$  must also belong to  $\mathcal{D}_m$ .

We will now prove that if a transaction set on ring-topology NoC is cyclic by definition in [9, 7], then it cannot be acyclic (thus it is cyclic) by the definition proposed in this paper. Let  $\mathcal{T}$  be a transaction set on ring-topology NoC that is cyclic by definition in [9, 7]. Assume by contradiction that  $\mathcal{T}$  is an acyclic transaction set by the definition proposed in this paper. Then the schedule of  $\mathcal{T}$  when all transactions have the same period can be generated in polynomial time using POBaseNoC. This, however, contradicts our proof in [7] that the problem of scheduling transactions in  $\mathcal{T}$  when all transactions have the same period is NP-complete.  $\square$