Real-time Scheduling of Concurrent Transactions in Multi-domain Ring Buses

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

Abstract—We address the problem of scheduling concurrent periodic real-time transactions on Multi-Domain Ring Bus (MDRB). The problem is challenging because although the bus allows multiple non-overlapping transactions to be executed concurrently, the degree of concurrency depends on the topology of the bus and of executed transactions. This is a different challenge compared to that of multi-processor real-time task scheduling. To solve this problem, first, we propose two novel efficient scheduling algorithms for *topographically-acyclic* transaction sets. The first algorithm is optimal for transaction sets under restrictive assumptions while the second one induces a competitive sufficient schedulable utilization bound for more general transaction sets. Then, we extend these two algorithms for the scheduling of *topographically-cyclic* transaction sets. Extensive simulations show that the proposed algorithm can schedule transaction sets with high bus utilization and is better than that of related works in most practical settings. The implementation of the algorithms in a real test-bed shows that they have relatively low execution-time overhead.

Index Terms—real-time communications, real-time scheduling, real-time Network-on-Chip scheduling.

1 Introduction

Demands for high performance computing systems have recently created significant interest in many-core System-on-Chip (SoC) both industry-wise and academic-wise [9], [10], [7]. In a many-core SoC, processing cores and other parts of the system are interconnected by a Network-on-Chip (NoC) [9], [10]. Since the performance of NoC can greatly affect the performance of the system as a whole, there have been many research efforts on NoC architectures [2]. Commercial many-core SoC with software-controlled NoC have also been developed. For example, the IBM Cell Broadband Engine processor (CellBE) [9] is well-distinguished for its high performance. The CellBE consists of twelve elements: a PowerPC core, eight Synergistic Processing Elements (SPE), a memory controller and two I/O controllers. All elements are connected through a high-speed ring bus.

Many-core SoC with software-controlled NoC is well suited for real-time applications. Consider highly critical real-time systems such as avionics and medical systems. A typical task in such systems executes the following activities: 1) collecting data from sensors that are tracking some physical events; 2) processing the data; 3) sending processed data or control signals to other tasks or actuators. An example can be a multipurpose status display task on an avionic system [18] which shows the status of all aircraft avionics devices. The task periodically gets data from I/O devices such as radars every dozen of milliseconds, then processes the data before sending information to a display task. A good implementation model for this software system on a many-core SoC is the thread streaming model [1]. In this model, different real-time tasks run on different processing elements. Real-time data transactions between tasks, I/O devices and main memory are executed through the NoC. The key idea is that since NoC accesses are software-controlled, software designers can schedule these data transactions deterministically.

In this paper, we study the real-time data transaction scheduling problem for a specific NoC architecture, the Multi-Domain Ring Bus (MDRB) (which is similar to what used in CellBE). In a MDRB, bus elements are connected through routers arranged in a ring configuration. Transactions that do not overlap (i.e., they are not routed through the same bus segment between routers) can be transferred concurrently. MDRB has been implemented in commercial systems [9], [3] and is a proven cost-effective high-performance solution for heterogeneous many-core SoC. The ring architecture requires smaller amount of wires compared to other common NoC architectures such as torus and mesh. As a consequence, this architecture is simpler and supports higher clock rates. At the same time, network utilization can be significantly higher compared to a simple shared bus because non-overlapping transactions can be transmitted in parallel.

However, exploiting such parallelism to maximize MDRB utilization while still providing strict real-time guarantees is challenging. At first glance, scheduling transactions in a MDRB bears similarity to parallel scheduling of tasks on multiprocessors, but there are major differences between the two problems. In particular, the degree of concurrency in a multiprocessor

depends on the number of processing elements. However, the degree of concurrency in a MDRB depends on the *topology* of the bus and executed transactions.

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 the PFair algorithm [5] and the Boundary-Fair algorithm [25], which are the optimal scheduling algorithms for multi-processors. Although this approach requires synchronization between bus elements 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 [11], a guaranteed-service NoC developed at Phillips Laboratory.

This research has two main contributions. First, we propose two novel scheduling algorithms POBase and POGen for real-time transaction sets which are *topographically acyclic*. These are transaction sets whose transaction overlaps do not create a cycle on the bus. POBase is optimal under a restrictive assumption on transactions' periods. Meanwhile, POGen induces a sufficient schedulable utilization bound for transaction sets without the restrictive assumption. We will show that the bound is highly competitive for typical MDRB implementations. Second, we propose cPOGen which extends POGen for topographically-cyclic transaction sets. To the best of our knowledge, our algorithms are the first *dynamic-priority* algorithms proposed for MDRB. Most previous works [23], [22], [16], [4] have focused on fixed-priority scheduling. Extensive simulations show that our approach allows much higher utilization for typical MDRB compared to the related works. The gain in performance of the proposed algorithm results in a higher algorithm overhead. However, we will show in our implementation that the overhead is relatively small in typical system settings.

The paper is organized as follows. We survey related works in Section 2. Section 3 defines the real-time bus transactions and the scheduling model. In Section 4, we discuss our proposed real-time scheduling algorithms for MDRB starting from a simpler case, that of topographically-acyclic transactions. Section 5 extends the algorithms to cyclic transaction sets. Section 6 provides a simulation-based evaluation of the proposed algorithms, while Section 7 discusses our implementation of the algorithms in a real system. We conclude our paper in Section 8.

2 Related Works

Many of the early works on hard real-time communication [15], [24], [20] 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. A system with multiple buses is considered in [12]. 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 problem in these works. The many-core SoC 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 focused on real-time communication on multi-domain buses. Most of these works [22], [23], [4], [16], [19] are concerned with the *fixed-priority* scheduling paradigm. For example, in [22], [23], 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. Although our work has the same assumption about multi-domain buses, our proposed scheduling algorithm is based on the *dynamic-priority* scheduling paradigm. To the best of our knowledge, our research is the first to do so. As will be shown in the evaluation section, the performance of our approach on a typical MDRB is better than that of related works.

A preliminary version of this work has appeared in [8]. Compared to [8], this paper has following improvements. First, we modified both algorithm POBase and POGen. The modification result in algorithms with significant lower time complexity than that of [8]. Second, we developed a complete solution for topographically-cyclic transaction sets which is not in [8]. Third, we added an evaluation section where we extensively compared the performance of the proposed algorithms with the state of the arts. Finally, we have fully implemented the algorithms in a real system and performed experiments to measure the algorithm overhead.

3 Real-time Bus Transaction and Scheduling Model

A model of the Multi-Domain Ring Bus (MDRB), in which we are interested, is shown in Figure 1. Our research focuses on MDRB with one bidirectional ring since they are most common; however, the proposed scheduling algorithms can also be applied to MDRB with two unidirectional rings (one clockwise and one counterclockwise) by analyzing each direction separately. In the MDRB shown in Figure 1, each bus element is connected to the MDRB through a dedicated bus router that is able to transmit a transaction toward its destination. The MDRB has a ring structure in which each router has direct connections through two bus segments to its two neighboring routers.

We study systems where applications running on multiple, possibly heterogeneous processing cores (i.e.

bus elements) exchange data through a MDRB. A *data* transaction is defined as a request made by an application to transfer a certain amount of data between two bus elements. We consider a scheduling problem where applications request *periodic* data transactions, each comprising an infinite sequence of jobs. Each data transaction is divided at the router level into multiple fixed-size packets called *atomic* transactions, which are then transferred hop-by-hop from the source to the destination. We do not impose specific constraints on the way routers and bus segments between them are implemented: worm-hole routers [21] are particularly suitable in networks-on-chip, but store-and-forward implementations are also possible. We assume that each data transaction has a fixed route which consists of bus elements through which it reaches the destination. Two data transactions *overlap* and can not be transferred concurrently if their routes share a same bus segment (e.g., each bidirectional bus segment can only be accessed in one direction at a time). However, multiple non-overlapping data transactions can be sent at the same time.

Our selection of the bus scheduling model aims at providing software designers with an accurate view of the bus real-time performance, while abstracting away the details of the low-level physical implementation. To this end, the proposed bus scheduling model does not schedule data transfers in term of atomic transactions. Instead, the bus elements which are the endpoints of a data transaction are synchronized and programmed to transfer a fixed portion of the data transaction at a time. Each portion consists of multiple atomic transactions. We call this portion of data a *unit* transaction. All unit transactions have an equal transmission time that is a *slot*, and each data transaction typically requires multiple slots (i.e. it is composed of multiple unit transactions). All scheduling decisions are made on a slot-by-slot basis.

Our model effectively abstracts most low-level implementation details because of two main reasons. 1) Since overlapping data transfers are never executed simultaneously, data packets (i.e. atomic transactions) never contend for access to the bus. Therefore, the schedule of atomic transactions is entirely predictable rather than being dictated by specific low-level atomic arbitration algorithms and/or router switching techniques. 2) The variable end-to-end delay required to transfer an atomic transaction from source to destination is hidden by the slot abstraction. In particular, the transmission time of a unit transaction includes the maximum transmission delay between any two elements. The rest of this section introduces the notation, terminology and basic results that will be used in the scheduling algorithms of Sections 4, 5.

3.1 Data Transaction Model

Let bus elements be indexed with a unique number in $[1, N^{\epsilon}]$ where N^{ϵ} is the number of bus elements.

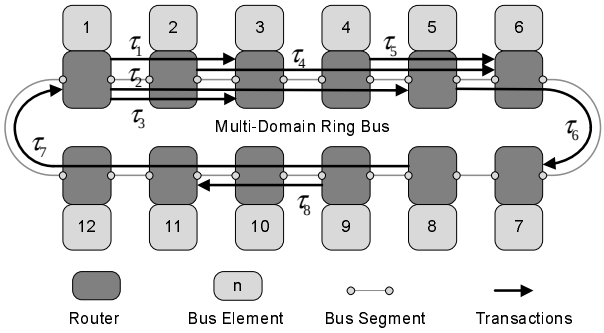

Fig. 1. Bus Architecture and Acyclic transaction set

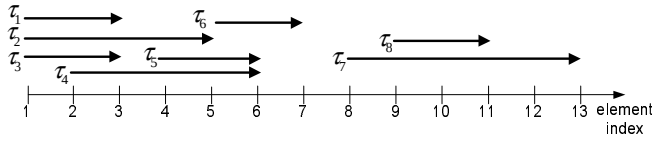

Fig. 2. Indexed straight line representation

We define $\mathcal{T}$ as the set of data transactions: $\mathcal{T} = \{\tau_i : i = [1, N]\}$. A data transaction τ_i is characterized by a tuple $\tau_i = (e_i, p_i, \epsilon_i^1, \epsilon_i^2)$ where e_i is the time that the bus spends to transmit a job of τ_i, p_i is the period of τ_i. Each job must complete within its period, i.e. relative deadlines are equal to periods. ϵ_i^1 and ϵ_i^2, where $\epsilon_i^1 \neq \epsilon_i^2$, are the indexes of the source and destination bus elements of the transactions. ϵ_i^1, ϵ_i^2 are called the *first* and *second* endpoint of τ_i, respectively. A transaction has two endpoints ϵ_i^1 and ϵ_i^2 if its route uses all consecutive routers from element ϵ_i^1 to ϵ_i^2 in the clockwise direction. Transaction τ_i is said to *go through* element ϵ if $\epsilon \neq \epsilon_i^1$, $\epsilon \neq \epsilon_i^2$ and element ϵ is on the route of τ_i. The bus utilization u_i of τ_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 $\mathcal{T}$.

Two transactions are said to *overlap* and can not be transferred concurrently on the bus if they use a same bus segment. 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 τ_i and τ_j overlap, and 0 otherwise. Figure 1 shows a transaction set where τ_1, τ_2, τ_3, and τ_4 overlap each other but they do not overlap τ_7.

A *pairwise overlap set* (PO-set) $\mathcal{D}$ is defined as a maximal subset of $\mathcal{T}$ such that $\forall \tau_i, \tau_j \in \mathcal{D} : OV(\tau_i, \tau_j) = 1$. For convenience, we consider that a transaction that does not overlap any transactions belongs to a PO-set that contains only that transaction. In general a transaction may belong to more than one PO-set. Figure 1 shows an example of a transaction set with four PO-sets: $\mathcal{D}_1 = \{\tau_1, \tau_2, \tau_3, \tau_4\}$, $\mathcal{D}_2 = \{\tau_2, \tau_4, \tau_5\}$, $\mathcal{D}_3 = \{\tau_4, \tau_5, \tau_6\}$, $\mathcal{D}_4 = \{\tau_7, \tau_8\}$. Let the total number of PO-sets in a transaction set be $N^{\mathcal{D}}$. Since each PO-set contains at least one element different from those of other PO-sets and transactions are arranged in an one dimensional

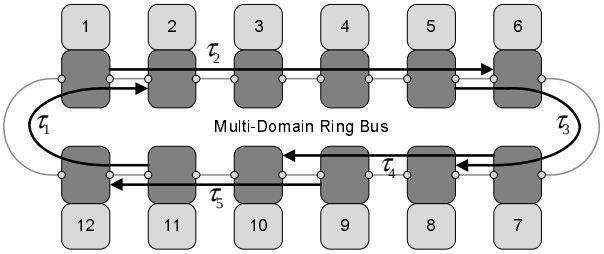

Fig. 3. Cyclic transaction set

space, $N^{\mathcal{D}} \leq N$.

A transaction set is said to be *acyclic* if there exists a bus element which has no transaction going through. The transaction set is *cyclic*, otherwise. Figure 1 shows an example of an acyclic where element 1, 7, and 8 have no transaction going through, whereas Figure 3 shows an example of a cyclic transaction set. For ease of identifying the first and second endpoints in the figure, we depict each transaction τ_i as an arrow which always directs from the first endpoint to the second endpoint of τ_i. The direction of the arrow does not imply the direction of the transaction.

3.2 Scheduling Model

We adopt the discrete scheduling model of PFair scheduling used in [5]. In this model scheduling decisions are made at integral values, starting from 0. The real interval between time $t \in \mathbb{N}$ and time $t+1$ i.e. $[t, t+1)$ is called *slot* t. We assume that every transaction's execution time and period are multiples of slots. Thereafter, we will use a slot as a time unit unless specified otherwise. A schedule S is defined as a function S: $\Gamma \times \mathbb{N} \mapsto \{0, 1\}$ where $S(\tau_i, t) = 1$ if and only if τ_i is scheduled at slot t. A schedule S is *valid* if and only if according to S, it *never* happens that a transaction is scheduled in the same slot together with one or more other transactions that overlap with it.

Given the constraint on overlapping transactions, a necessary condition on the schedulability of a transaction set can be easily derived as in Theorem 3.1.

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

$$\forall \mathcal{D} \subset \mathcal{T} : u^{\mathcal{D}} = \sum_{\forall \tau_i \in \mathcal{D}} u_i \leq 1 \qquad (3.1)$$

Proof: Since, by definition, no two transactions of a PO-set $\mathcal{D}$ can be scheduled concurrently, all transactions of $\mathcal{D}$ must be scheduled in sequence. In other words, the transactions of $\mathcal{D}$ can be considered to be sharing one resource. Therefore, Inequality 3.1 must be satisfied. □

Let $E(\epsilon_k)$ be a set of all transactions in $\mathcal{T}$ that use the bus segment between bus element ϵ_k and $\epsilon_k + 1$. The following lemma is necessary for later discussion.

Lemma 3.1: Given a transaction set $\mathcal{T}$ that satisfies the necessary condition, the following inequality holds.

$$\sum_{\forall \tau_i \in E(\epsilon_k)} u_i \leq 1$$

Proof: Since transactions in $E(\epsilon_k)$ pairwise overlap, there exists $\mathcal{D}$ such that $E(\epsilon_k) \subseteq \mathcal{D}$. Therefore the lemma is implied by Theorem 3.1. □

3.3 Indexed Straight-line Representation of Acyclic Transaction Sets

For ease of presentation, thereafter, we use the indexed straight-line representation described below to model acyclic transaction sets. Given an acyclic transaction set, we select a bus element which has no transaction going through to be the *first element*. Then, the bus elements are indexed ascendingly from 1 to N^{ϵ} in clockwise direction in which the first element's index is 1. Bus elements in Figure 1 are indexed following this definition. Since there are no transaction going through element 1, the overlaps between transactions in the acyclic transaction set remains the same if we do the following transformation: 1) let bus element N^{ϵ}, instead of connecting to bus element 1, connect to an additional bus element which is indexed $N^{\epsilon}+1$; 2) change every transaction which has the second endpoint at 1, i.e. $\tau_i = (e_i, p_i, \epsilon_i^1, 1)$, to be $\tau_i = (e_i, p_i, \epsilon_i^1, N^{\epsilon}+1)$. For example, in Figure 1, $\tau_7 = (e_7, p_7, 8, 1)$ will be changed to be $\tau_7 = (e_7, p_7, 8, 13)$. Since the overlaps between transactions are still the same after the transformation, a valid schedule of the transformed transaction set is also a valid schedule of the original transaction set and vice versa. Given this transformation, the acyclic transaction set can be represented as a set of overlapping line intervals on an indexed straight line where each line interval corresponds to a transaction and the straight line is indexed from 1 to $N^{\epsilon}+1$. Figure 2 shows the indexed straight line representation of the transaction set shown in Figure 1. The following properties are obvious in the straight-line representation of an acyclic transaction set.

Property 1: For every transaction τ_i, we have $\epsilon_i^1 < \epsilon_i^2$.

Property 2: For every transaction τ_i and τ_j, τ_i and τ_j overlap if and only if $\epsilon_i^1 < \epsilon_j^2$ and $\epsilon_j^1 < \epsilon_i^2$.

We study the scheduling problem for acyclic transaction sets in Section 4. We then extend our solution to cyclic transaction sets in Section 5.

4 Scheduling Algorithms for Acyclic Transactions

In this section we present our scheduling algorithms for the proposed real-time transaction sets on the ring buses. The discussion is divided into two parts.

First, we propose an algorithm, namely POBase, which schedule every acyclic transaction set whose transactions have the *same* period. We will prove that the necessary condition (Theorem 3.1) is also the sufficient condition for same-period acyclic transaction set to be schedulable by POBase. Therefore, POBase is optimal for these transaction sets.

Second, a scheduling algorithm, namely POGen, is proposed to schedule acyclic transaction sets whose

transactions do not have the same period. POGen, which is built based on POBase, is an online algorithm. POGen can schedule all transaction sets whose PO-set utilizations satisfy the following utilization bound:

$$\forall \mathcal{D} \subset \mathcal{T} : u^{\mathcal{D}} \leq \frac{\mathsf{L}-1}{\mathsf{L}}, \tag{4.1}$$

where L is defined as the greatest common divisor of all transaction periods. Although the utilization bound is sufficient, it approximates 1 when L is large. We believe that this assumption holds in most practical real-time applications [18]. As we will show in the implementation section, with the speed of the state of the art many-core SoC [3], the practical slot size is about $100us$ to $10us$ (which is also the size of a time unit in our definition). Meanwhile, the period granularity in practical real-time applications [18] is at the level of milliseconds. That means L has practical values ranging from 10 to 100 slots. This results in the utilization bound between 0.9 and 0.99.

4.1 The POBase algorithm

The problem of acyclic same-period transaction set scheduling is similar to the Interval Coloring Problem [14]. More specifically, the latter is a special case of the former because it assumes that all transactions have the same execution time. Hence, the coloring algorithm in [14] can only handle this special case. Our proposed algorithm POBase is a new algorithm to solve the problem at hand. POBase is a first-fit algorithm with respect to a transaction ordering. More specifically, in POBase, the transactions are ordered ascendingly by their first endpoint (stored in list $\mathcal{L}$). Then, each transaction in $\mathcal{L}$ is assigned to the earliest slots where no smaller-ordered overlapping transaction has been already assigned to[1]. This condition is enforced by the use of array lastEndpoint (Step 6). lastEndpoint has size equal to the transactions' period p. The initial values of all items of lastEndpoint is 1 which is also the smallest index of the bus elements. Except for the initial value, during the algorithm execution, the value of item t of lastEndpoint will be the second endpoint (Step 8) of the last transaction that has been assigned to slot t. Since transactions are being assigned in ascending order of their first endpoints, if condition $\mathsf{lastEndpoint}[t] \leq \epsilon_i^1$ in Step 6 is satisfied then τ_i does not overlap with all transactions that have been assigned to t before τ_i. We will formally prove this statement in Lemma 4.2. This proof requires Lemma 4.1. Finally, the proof of POBase's optimality will be shown in Theorem 4.1.

Figure 4 shows an example of the schedule generated by POBase for the transaction set shown in Figure 1 whose transactions have period equal to 8 and execution times: $e_1 = 2, e_2 = 1, e_3 = 2, e_4 = 3, e_5 = 4, e_6 = 1, e_7 = 4, e_8 = 4$. Consider the schedule of transactions

1. The transactions can also be ordered by their second endpoint and the schedule is generated in descending ordered of the order list.

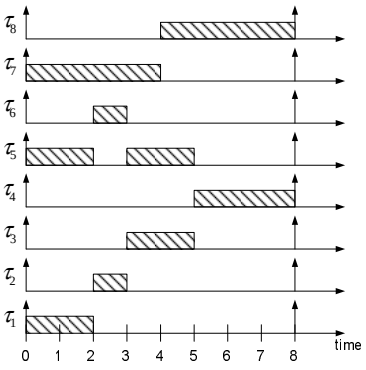

Fig. 4. An example of the POBase algorithm

of $\mathcal{D}_2 = \{\tau_2, \tau_4, \tau_5\}$. τ_5 is scheduled in slots $\{0, 1, 3, 4\}$, because its smaller-ordered overlapping transactions τ_2 and τ_4 are scheduled in slots $\{2, 5, 6, 7\}$.

Lemma 4.1: At each iteration of Step 6, for every τ_j which has been assigned to slot t, $\epsilon_j^2 \leq \mathsf{lastEndpoint}[t]$.

Proof: We prove by induction.

Base case: Consider the first iteration, the lemma holds because there is not any transaction being assigned.

Induction case: Assume that the lemma holds at iteration k where τ_j is being assigned, we will prove that it also holds at iteration $k+1$. Consider slot t to which τ_j is assigned. Due to the condition at Step 6, we have $\mathsf{lastEndpoint}[t] \leq \epsilon_j^1$. Then by the induction assumption, we have: for every τ_l that has been assigned to t before iteration k, $\epsilon_l^2 \leq \epsilon_j^1$. Furthermore, when τ_j is assigned to duration t, we have $\mathsf{lastEndpoint}[t] = \epsilon_j^2$ after Step 8. Since by Property 1 of the indexed straight-line presentation, $\epsilon_j^1 < \epsilon_j^2$, we have the lemma holds at iteration $k+1$. □

Algorithm 1 POBase

Input: $\mathcal{T}$ such that $\forall \tau_i \in \mathcal{T} : p_i = p$
Output: schedule S for period p

```
1:  L ← list of ∀τ_i ∈ T in ascending order of ε_i^1
2:  ∀t ∈ [0, p) : lastEndpoint[t] ← 1
3:  for each τ_i ∈ L do
4:      r ← e_i
5:      for each t ∈ [0, p) do
6:          if lastEndpoint[t] ≤ ε_i^1 then
7:              S(τ_i, t) ← 1
8:              lastEndpoint[t] ← ε_i^2
9:              r ← r − 1
10:             if r = 0 then
11:                 break //complete schedule assignment of τ_i
```

Lemma 4.2: Slot t has not been assigned to any overlapping transaction of τ_i if and only if $\mathsf{lastEndpoint}[t] \leq \epsilon_i^1$.

Proof:

Necessary condition: We prove this condition by showing that if $\mathsf{lastEndpoint}[t] > \epsilon_i^1$ then slot t has been assigned to a transaction that overlaps τ_i. Since

$\mathsf{lastEndpoint}[t] > \epsilon_i^1$, there must exist a transaction τ_j that has been assigned to t before τ_i where $\epsilon_j^2 = \mathsf{lastEndpoint}[t] > \epsilon_i^1$. And since τ_j has been assigned to the slot before τ_i, $\epsilon_i^1 \geq \epsilon_j^1$. Therefore, we have: $\epsilon_j^2 > \epsilon_i^1$ and $\epsilon_i^2 > \epsilon_j^1$. Then, by Property 2 of the indexed straight-line presentation, τ_i and τ_j overlap.

Sufficient condition: If $\mathsf{lastEndpoint}[t] \leq \epsilon_i^1$, then by Lemma 4.1 we have that: for every τ_j that has been assigned to slot t before τ_i, $\epsilon_j^2 \leq \mathsf{lastEndpoint}[t] \leq \epsilon_i^1$. Therefore, by Property 2 of the indexed straight-line presentation, τ_j and τ_i do not overlap. □

Theorem 4.1: POBase is optimal for same-period acyclic transaction sets.

Proof: The generated schedule is valid because a transaction is not scheduled in the same slot with its overlapping transactions (Lemma 4.2) . It remains to show that if a transaction set satisfies the necessary condition, then at the end of the algorithm,

$$\forall \tau_i : \sum_{x \in [0,p)} S(\tau_i, x) = e_i. \tag{4.2}$$

We will prove this by induction.

Base case: Consider the first iteration of the for-loop starting at Step 3. In this iteration, the schedule of τ_1 in $\mathcal{L}$ is generated. Since all items of lastEndpoint have value 1, the condition at Step 6 satisfies for every t. Furthermore, we have $e_1 \leq p$. Therefore, at the end of the iteration, Equation 4.2 must holds for τ_1 i.e. $\sum_{x \in [0,p)} S(\tau_1, x) = e_1$.

Induction case: Assume after iteration k of the for-loop starting at Step 3, Equation 4.2 holds for all transactions $\{\tau_i : i \in [1,k]\}$. We will prove that Equation 4.2 also holds for τ_{k+1} after iteration $k+1$. By contradiction, assume that at the end of the iteration $k+1$, $\sum_{x \in [0,p)} S(\tau_{k+1}, x) < e_{k+1}$. Let $E(\epsilon_{k+1}^1)$ be the set of transactions that use the bus segment between bus element ϵ_{k+1}^1 and $\epsilon_{k+1}^1 + 1$. By the way the transactions are ordered and Property 2 of the indexed straight-line presentation, we have that $\forall \tau_i \in \mathcal{T}$ if the schedule of τ_i has been generated before τ_{k+1} and τ_i overlaps τ_{k+1} then $\epsilon_i^1 \leq \epsilon_{k+1}^1 < \epsilon_i^2$. Therefore $\tau_i \in E(\epsilon_{k+1}^1)$. It is because $\epsilon_i^1 \leq \epsilon_{k+1}^1 < \epsilon_i^2$. In other words, among all the transactions that overlap with τ_{k+1}, only transactions in $E(\epsilon_{k+1}^1)$ have their schedule been generated. Therefore, the contradiction assumption occurs only when:

$$\sum_{\tau_i \in E(\epsilon_{k+1}^1)} \sum_{x \in [0,p)} S(\tau_i, x) = p. \tag{4.3}$$

Since the following is true:

$$\forall \tau_i \in E(\epsilon_{k+1}^1) \setminus \{\tau_{k+1}\} : \sum_{x \in [0,p)} S(\tau_i, x) \leq e_i,$$

by the contradiction assumption and Equation 4.3 we have: $\sum_{\tau_i \in E(\epsilon_{k+1}^1)} e_i > p$. This contradicts with Lemma 3.1 which implies that $\sum_{\tau_i \in E(\epsilon_{k+1}^1)} e_i \leq p$. Therefore, at the end of the iteration, Equation 4.2 must hold for τ_{k+1}. This completes the proof. □

POBase Analysis: an efficient sorting algorithm has time complexity $O(N \log(N))$. Furthermore, Step 6 to 11 requires constant number of operations. Therefore the time complexity of POBase to build a schedule of p slots for N transactions is $O(N * \max(\log(N), p))$.

4.2 The POGen algorithm

In this subsection we propose an online scheduling algorithm (POGen) for acyclic transaction sets whose transactions do not have the same period. In POGen, the execution time line from 0 to the hyper-period h, i.e. $[0, h)$, is divided into a set of consecutive *scheduling intervals*: $\{\mathsf{int}^k = [t^k, t^{k+1}) : k \in \mathbb{N} \wedge 0 \leq t^k < t^{k+1} < h\}$. Let $|\mathsf{int}^k| = t^{k+1} - t^k$. In each scheduling interval int^k, each transaction τ_i is assigned an *interval load* l_i^k which is the number of slots in the interval allocated to schedule τ_i. The interval loads of each transaction is calculated such that at the end of each interval, the transaction's execution approximates its execution in the fluid scheduling model [13]. The interval load of a PO-set is the sum of the interval loads of its transactions. Given the interval loads of all transactions in interval int^k, POBase is used to generate the schedule of int^k. As shown in the previous subsection, the interval schedule given by POBase will be feasible if and only if:

$$\forall \mathcal{D} \subset \mathcal{T} : \sum_{\tau_i \in \mathcal{D}} l_i^k \leq |\mathsf{int}^k|.$$

A schedule of a transaction set, which is generated by POGen, is feasible if it satisfies the following two conditions:

Condition 1: for each transaction τ_i, the sum of the interval loads over the transaction period is equal to e_i.

Condition 2: there is a feasible schedule for every scheduling interval.

In the following paragraphs, we will discuss our solution to identify the scheduling intervals and the interval loads which induces a feasible schedule.

Our proposed solution is inspired by the work in [25]. However, since this work does not have the transaction overlap assumption, their proposed algorithms can not be used for the problem at hand. In POGen, scheduling intervals must respect two fundamental properties: 1) the arrival time (also deadline) of any transaction must coincide with the finishing time of a scheduling interval and the start time of the next one; 2) the minimum length of any scheduling interval must be at least L where L is the greatest common divisors of all transaction periods. As we will show in Theorem 4.2, the second property is essential to induce a feasible utilization bound: intuitively, longer scheduling intervals allow POGen to better approximate the fluid scheduling model. There are multiple feasible assignments of scheduling intervals that respects the two properties. For example, Figure 6 shows the scheduling intervals induced by the set of three transactions shown in Figure 5, where $\tau_1 = \{e_1 = 1, p_1 = 2, \epsilon_1^1 = 1, \epsilon_1^2 = 3\}$,

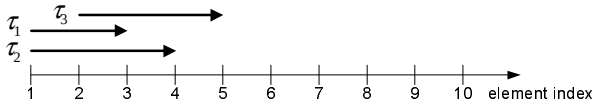

Fig. 5. Example of three transactions

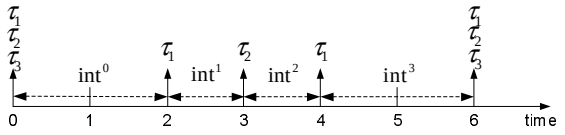

Fig. 6. Scheduling intervals on the execution time line

$\tau_2 = \{e_2 = 1, p_2 = 3, \epsilon_2^1 = 1, \epsilon_2^2 = 4\}$ and $\tau_3 = \{e_3 = 1, p_3 = 6, \epsilon_3^1 = 2, \epsilon_3^2 = 5\}$. In this example, the scheduling intervals are the intervals between two closest arrival times of any two transactions. Note that by definition of L and since all transactions arrive at time 0, it follows that the minimum length of scheduling intervals in the example is indeed L. An alternative feasible definition for scheduling intervals consists in assigning $t^k = k\mathsf{L}$, e.g. all scheduling intervals have fixed length L. By definition of L, it then follows that the arrival time of any transaction coincides with the start time t^k of some interval int^k. In the rest of this section, we will not restrict ourselves to any specific interval assignment, instead only assuming that scheduling intervals respect the two fundamental properties.

With regard to the interval loads, we define for each transaction τ_i and scheduling interval int^k a lag function:

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

The function calculates how much time τ_i must be executed in interval int^k such that at the end of int^k it is scheduled according to the fluid scheduling model [13]. We also define for each PO-set $\mathcal{D}$ a similar lag function:

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

The goal of POGen is to generate a *feasible load set* for each interval int^k, that is, a set of transaction loads that satisfy the following inequalities:

$$\forall \tau_i \in \mathcal{T} : \lfloor \mathsf{lag}(\tau_i, \mathsf{int}^k) \rfloor \leq l_i^k \leq \lceil \mathsf{lag}(\tau_i, \mathsf{int}^k) \rceil, \quad (4.4)$$

$$\forall \mathcal{D} \subset \mathcal{T} : \lfloor \mathsf{lag}(\mathcal{D}, \mathsf{int}^k) \rfloor \leq \sum_{\tau_i \in \mathcal{D}} l_i^k \leq \min(|\mathsf{int}^k|, \lceil \mathsf{lag}(\mathcal{D}, \mathsf{int}^k) \rceil). \quad (4.5)$$

Inequality 4.4 sets conditions on the interval load for each transaction, based on the closest integral values of the lag functions. Inequality 4.5 sets conditions on the total interval load of each PO-set. Note that the right side of Inequality 4.5 guarantees that each PO-set with feasible loads is schedulable in int^k by POBase, that is, Condition 2 is satisfied for int^k. Similarly, if all loads satisfy the lower bounds of Inequalities 4.4, then the generated schedule satisfies Condition 1. The reason is as follows. Consider the last scheduling interval of a period of transaction τ_i: $\mathsf{int} = [t, a * p_i)$ where t and a are some integers, the lag function of τ_i is:

$$\mathsf{lag}(\tau_i, \mathsf{int}) = a * u_i * p_i - \sum_{x \in [0,t)} S(\tau_i, x).$$

Since $u_i * p_i = e_i$ is an integer, and so is $S(\tau_i, x)$, $\lfloor \mathsf{lag}(\tau_i, \mathsf{int}) \rfloor = \mathsf{lag}(\tau_i, \mathsf{int})$. That means the total interval loads of τ_i up to slot $a * p_i$, which is calculated as:

$$\lfloor \mathsf{lag}(\tau_i, \mathsf{int}) \rfloor + \sum_{x \in [0,t)} S(\tau_i, x),$$

is equal to $a * e_i$ and satisfies Condition 1. However using only the lower bound loads does not guarantee that Inequality 4.5 can be satisfied at the same time. This is also true if only upper bound loads are used. The following example illustrates this point. Consider again example of the transaction set in Figure 5 and 6. If the algorithm runs with interval loads to be their lower bound loads, then the schedule of interval $[4, 6)$ is not feasible because the total load in this interval is 3. If otherwise, the upper bound loads are used only, then the schedule of interval $[0, 2)$ is also not feasible because the total load in this interval is 3. An algorithm that generates feasible schedules must use a combination of these values and computing this is not trivial.

POGen achieves this feat by iteratively computing a feasible load set for each scheduling interval. It is an online algorithm which is invoked at the beginning of each interval and generates the schedule for that interval. In Lemma 4.3, we first show that the following inequalities initially hold at the beginning of the first interval int^0:

$$\forall \mathcal{D} \subset \mathcal{T} : \lfloor \mathsf{lag}(\mathcal{D}, \mathsf{int}^k) \rfloor \leq |\mathsf{int}^k|, \quad (4.6)$$

$$\forall \mathcal{D} \subset \mathcal{T} : \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0, t^k)} S(\tau_i, x) \geq \lfloor u^{\mathcal{D}} * t^k \rfloor. \quad (4.7)$$

A feasible load set is then computed in Step 1 of POGen by the GenerateLoad procedure, which is assumed to honor the following proposition:

Proposition 4.1: Assume that all PO-sets satisfy the utilization bound in Inequalities 4.1. If Inequalities 4.6, 4.7 hold for int^k, then GenerateLoad computes a feasible load set for int^k.

Given a feasible load set for interval int^0, Lemma 4.3 guarantees that Inequalities 4.6, 4.7 again hold for int^1. Hence, in the next execution of POGen at t^1, GenerateLoad can compute a feasible load set for int^1, and so on and so forth for all scheduling intervals in the hyper-period. Since a feasible load set is obtained for all scheduling intervals, Condition 1 and Condition 2 are satisfied and thus POGen generates a feasible schedule of $\mathcal{T}$. In the next Section 4.3, we will prove that GenerateLoad indeed honors Proposition 4.1.

Algorithm 2 POGen

Input: transaction set $\mathcal{T}$, interval int^k
Output: schedule S for int^k

1: $\{l_i^k : \forall i \in [1, N]\} \leftarrow \mathsf{GenerateLoad}(\mathcal{T}, \mathsf{int}^k)$
2: $\mathcal{T}' \leftarrow \left\{\{l_i^k, |\mathsf{int}^k|, \epsilon_i^1, \epsilon_i^2\} : \forall i \in [1, N]\right\}$
3: S for $\mathsf{int}^k \leftarrow \mathsf{POBase}(\mathcal{T}')$

Lemma 4.3: If GenerateLoad honors Proposition 4.1, then Inequalities 4.6, 4.7 hold for every scheduling intervals.

Proof: We prove by induction.

Base step: Consider the first scheduling interval $\mathsf{int}^0 = [0, t^1)$. Inequalities 4.6 for this interval hold because

$$\forall \mathcal{D} \subset \mathcal{T} : \lfloor \mathsf{lag}(\mathcal{D}, \mathsf{int}^0) \rfloor = \lfloor u^{\mathcal{D}} * t^1 \rfloor \leq |\mathsf{int}^1|,$$

and Inequalities 4.7 hold because

$$\forall \mathcal{D} \subset \mathcal{T} : \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0,0)} S(\tau_i, x) = 0 = \lfloor u^{\mathcal{D}} * 0 \rfloor.$$

Induction step: Assume that Inequalities 4.6, 4.7 hold in every scheduling interval up to int^k. We prove that Inequalities 4.6, 4.7 also hold before the execution of GenerateLoad at interval int^{k+1}. Since Inequalities 4.6, 4.7 are satisfied at interval int^k, GenerateLoad generates a feasible load set and POBase generates a feasible schedule for the interval. Therefore after Step 3, we have:

$$\forall \mathcal{D} \subset \mathcal{T} : \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [t^k, t^{k+1})} S(\tau_i, x) = \sum_{\tau_i \in \mathcal{D}} l_i^k.$$

Then by the left side of Inequalities 4.5, we obtain the following which proves that Inequalities 4.7 hold for int^{k+1}.

$$\begin{aligned}
&\forall \mathcal{D} \subset \mathcal{T} : \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0, t^{k+1})} S(\tau_i, x) \\
&= \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0, t^k)} S(\tau_i, x) + \sum_{\tau_i \in \mathcal{D}} l_i^k \\
&\geq \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0, t^k)} S(\tau_i, x) + \\
&\left\lfloor u^{\mathcal{D}} * t^{k+1} \right\rfloor - \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0, t^k)} S(\tau_i, x) \\
&= \lfloor u^{\mathcal{D}} * t^{k+1} \rfloor
\end{aligned}$$

Now consider Inequalities 4.6. Notice that since $S(\tau_i, x)$ is integer, we have:

$$\begin{aligned}
&\forall \mathcal{D} \subset \mathcal{T} : \lfloor \mathsf{lag}(\mathcal{D}, \mathsf{int}^{k+1}) \rfloor \\
&= \left\lfloor u^{\mathcal{D}} * t^{k+2} \right\rfloor - \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0, t^{k+1})} S(\tau_i, x).
\end{aligned}$$

Since Inequalities 4.7 hold for int^{k+1}, Inequalities 4.6 also hold because:

$$\begin{aligned}
&\forall \mathcal{D} \subset \mathcal{T} : \lfloor \mathsf{lag}(\mathcal{D}, \mathsf{int}^{k+1}) \rfloor \\
&= \left\lfloor u^{\mathcal{D}} * t^{k+2} \right\rfloor - \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0, t^{k+1})} S(\tau_i, x) \\
&\leq \lfloor u^{\mathcal{D}} * t^{k+2} \rfloor - \lfloor u^{\mathcal{D}} * t^{k+1} \rfloor \\
&\leq \lceil u^{\mathcal{D}} * (t^{k+2} - t^{k+1}) \rceil \leq |\mathsf{int}^{k+1}|.
\end{aligned}$$

This completes the proof. □

4.3 The GenerateLoad procedure

As we mentioned, procedure GenerateLoad searches for a feasible load set of each scheduling interval. 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 at hand is equivalent to the problem of circulations in graphs with loads and lower bounds [14]. This is the problem of finding a feasible circulation flow in a directed graph where 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 4.1, 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 intuitively describe the construction of a directed graph from the input of GenerateLoad. Each vertex of the constructed graph represents a PO-set $\mathcal{D}_j$. We define for each vertex, a *PO-set edge* $g_j^{\mathcal{D}}$ whose real-valued flow $f_j^{\mathcal{D}}$ represents the interval load of the corresponding PO-set. An integer-valued lower bound $b_j^{\mathcal{D}}$ and an integer-valued capacity $c_j^{\mathcal{D}}$ are defined for each of the PO-set edges such that Inequalities 4.5 are imposed on their flow values, i.e.:

$$\forall \mathcal{D}_j \subset \mathcal{T} : b_j^{\mathcal{D}} \leq f_j^{\mathcal{D}} \leq c_j^{\mathcal{D}}. \tag{4.8}$$

where

$$\begin{aligned}
b_j^{\mathcal{D}} &= \lfloor \mathsf{lag}(\mathcal{D}_j, \mathsf{int}^k) \rfloor, \\
c_j^{\mathcal{D}} &= \min(|\mathsf{int}^k|, \lceil \mathsf{lag}(\mathcal{D}_j, \mathsf{int}^k) \rceil).
\end{aligned}$$

Furthermore, for each transaction τ_i, a *transaction edge* is defined whose real-valued flow f_i represents the interval load of the corresponding transaction. A lower bound value b_i and a capacity c_i are defined for each of the transaction edges such that Inequalities 4.4 are imposed on their flow values:

$$\forall \tau_i \in \mathcal{T} : b_i = \lfloor \mathsf{lag}(\tau_i, \mathsf{int}^k) \rfloor \leq f_i \leq c_i = \lceil \mathsf{lag}(\tau_i, \mathsf{int}^k) \rceil. \tag{4.9}$$

The flow of a transaction edge entering a vertex represents the contribution of the corresponding transaction's interval load to the corresponding PO-set's interval load. The endpoints and the direction of each edge are defined in such a way that the values of the flows in and out a vertex preserve the relationship between

the interval load of the corresponding PO-set and that of its transactions. The graph has a feasible circulation flow which represents a feasible load set.

The following definition is necessary for the graph construction. Let the *index PO-set order* of a transaction set $\mathcal{T}$ be an ordered list of all PO-sets in $\mathcal{T}$ where PO-set $\mathcal{D}$ with smaller $\min_{\tau_i \in \mathcal{D}_j} \epsilon_i^2$ has smaller index. Ties are broken arbitrarily. Since each PO-set has only one value $\min_{\tau_i \in \mathcal{D}_l} \epsilon_i^2$, the order is well-defined. The transaction set in Figure 1 has the index PO-set order be $\{\mathcal{D}_j : j \in [1,4]\}$ where $\mathcal{D}_1 = \{\tau_1, \tau_2, \tau_3, \tau_4\}$, $\mathcal{D}_2 = \{\tau_2, \tau_4, \tau_5\}$, $\mathcal{D}_3 = \{\tau_4, \tau_5, \tau_6\}$, $\mathcal{D}_4 = \{\tau_7, \tau_8\}$. Figure 7 shows the graph G constructed from the transaction set in Figure 1. Transaction edges are represented by solid lines while PO-set edges are represented by dotted lines.

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

- For each PO-set $\mathcal{D}_j$ in the index PO-set order, define a vertex v_j.
- For each PO-set $\mathcal{D}_j$ in the index PO-set order, define a directed edge $g_j^{\mathcal{D}}$ with capacity $c_j^{\mathcal{D}} = \min(\lfloor \mathsf{int}^k \rfloor, \lceil \mathsf{lag}(\mathcal{D}_j, \mathsf{int}^k) \rceil)$ and lower bound $b_j^{\mathcal{D}} = \lfloor \mathsf{lag}(\mathcal{D}_j, \mathsf{int}^k) \rfloor$. Let $g_j^{\mathcal{D}}$ be a *PO-set edge*.
- For each transaction τ_i, define a directed edge g_i with capacity $c_i = \lceil \mathsf{lag}(\tau_i, \mathsf{int}^k) \rceil$, and lower bound $b_i = \lfloor \mathsf{lag}(\tau_i, \mathsf{int}^k) \rfloor$. Let g_i be a *transaction edge*.
- $\{g_i : \tau_i \in \mathcal{D}_1\}$ are edges that enter v_1; $g_1^{\mathcal{D}}$ is an edge that exits v_1.
- $\forall j : 1 < j \leq N^{\mathcal{D}}$, $\{g_i : \tau_i \in \mathcal{D}_j \setminus \mathcal{D}_{j-1}\}$ and $g_{j-1}^{\mathcal{D}}$ are edges that enter v_j; $\{g_i : \tau_i \in \mathcal{D}_{j-1} \setminus \mathcal{D}_j\}$ and $g_j^{\mathcal{D}}$ are edges that exits v_j. This construction step deals with the situation where two PO-sets $\mathcal{D}_{j-1}, \mathcal{D}_j$ share some transactions. Intuitively, to preserve the relationship between the interval loads of the PO-sets and that of its transactions, the transaction edge of a transaction common to two PO-sets would have to enter the two corresponding vertexes v_{j-1}, v_j. Since in a qualified graph, each directed edge can enter at most one vertex, this situation must be avoided. This can be accomplished by representing the interval loads of the common transactions on the second PO-set (v_j) as the interval load of the first PO-set (i.e., $g_{j-1}^{\mathcal{D}}$ enters v_j) minus the interval load of the transactions that are only in the first set (i.e., $\{g_i : \tau_i \in \mathcal{D}_{j-1} \setminus \mathcal{D}_j\}$ exit v_j). Lemma 4.4 will detail the proof of this argument.
- $V = \{v_j : j \in [1, N^{\mathcal{D}}]\}$
- $E = \{g_i : j \in [1, N]\} \cup \{g_j^{\mathcal{D}} : j \in [1, N^{\mathcal{D}}]\}$

Finally, the graph flow is subject to the flow conversation constraint [14] in which given a vertex, the sum of the flow values entering it minus the sum of the flow values existing it is zero. As a graph construction example, consider PO-set $\mathcal{D}_2$. Vertex v_2 has an output PO-set edge $g_2^{\mathcal{D}}$ which represents the interval load of $\mathcal{D}_2$.

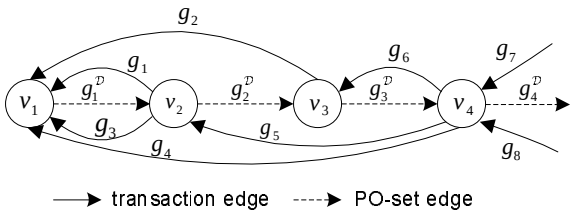

Fig. 7. Constructed graph G

Since $\mathcal{D}_1$ has τ_2 and τ_4 in common with $\mathcal{D}_2$ but not τ_1 and τ_3, v_2 has an input PO-set edge $g_1^{\mathcal{D}}$ which represents the interval load of $\mathcal{D}_1$ and two output transaction edges g_1 and g_3 that represent the interval loads of τ_1 and τ_3, respectively. Furthermore, note that edges g_2 and g_4 for the common transactions τ_2 and τ_4 enter v_1, not v_2. g_2 exits v_3, but g_4 exists v_4 because τ_4 also belongs to $\mathcal{D}_3$. Finally v_2 has an input transaction edge g_5 that represents the interval load of τ_5. Since τ_5 is in $\mathcal{D}_2$ and $\mathcal{D}_3$ but not $\mathcal{D}_1$ and $\mathcal{D}_4$, g_5 exits v_4. Lemma 4.4 shows that G is indeed a directed graph.

Lemma 4.4: G is a directed graph.

Proof: Since every edge of G is directed, it remains to show that each edge has only one or two endpoints. There is one edge defined for each PO-set and one edge defined for each transaction.

For each PO-set $\mathcal{D}_j$, the PO-set edge g_j^D exits only v_j. In addition, g_j^D enters only v_{j+1} when $j < N^{\mathcal{D}}$. Therefore each PO-set edge exists exactly one vertex and enters at most one vertex.

By the index PO-set ordering, if $\tau_i \in \mathcal{D}_j \setminus \mathcal{D}_{j-1}$, then $\tau_i \notin \mathcal{D}_k \setminus \mathcal{D}_{k-1}$ where $j < k \leq N^{\mathcal{D}}$. Therefore, the elements of the following set are disjoint: $\mathcal{A} = \big\{\{g_i : \tau_i \in \mathcal{D}_1\}, \{g_i : \tau_i \in \mathcal{D}_j \setminus \mathcal{D}_{j-1}\} : j \in (1, N^{\mathcal{D}}]\big\}$. By definition, $\mathcal{A}$ contains the transaction edges of G that enter some vertices. Also the union of the elements of $\mathcal{A}$ is $\{g_i : \tau_i \in \mathcal{T}\}$. Therefore, each transaction edges enters exactly one vertex.

By a similar proving technique, we can show that each transaction edge exists at most one vertex. Due to space constraints, we skip the detailed proof. In conclusion, every edge of G has at most two endpoints and is directed. □

It remains to show that GenerateLoad honors Proposition 4.1 and therefore POGen generates feasible schedules for all transaction sets that satisfy the utilization bound of Inequalities 4.1. For simplicity of exposition, we split the proof in multiple lemmas. First, Lemma 4.5 proves an important property of graph G regarding the flow values. Then, this property will be used to prove in Lemma 4.6 that graph G has a feasible flow if Inequalities 4.6, 4.7 are satisfied for interval int^k and furthermore all PO-sets satisfy an utilization constraint based on $\lfloor \mathsf{int}^k \rfloor$. 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 4.7). Finally, we will show that the utilization bound of Inequalities 4.1 implies the utilization bound used in Lemma 4.6. Hence, Proposition 4.1 holds.

Lemma 4.5: A flow in graph G honors the flow conservation constraint at every vertex v_j if and only if the following equalities hold for every PO-set $\mathcal{D}_j$:

$$\sum_{\tau_i \in \mathcal{D}_j} f_i = f_j^{\mathcal{D}}. \tag{4.10}$$

Proof: We prove this by induction over the ordered set of vertices.

Base case: By the graph construction rules regarding the edges that enter and exist v_1, the flow conservation constraint holds at vertex v_1 if and only if:

$$\sum_{\tau_i \in \mathcal{D}_1} f_i = f_1^{\mathcal{D}}$$

Induction case: Assume that the lemma holds for every PO-set from $\mathcal{D}_1$ to $\mathcal{D}_{j-1}$. We will prove that the lemma also holds for $\mathcal{D}_j$. By the graph construction rules regarding the edges that enter and exist v_j and the definition of the flow conservation constraint at vertex v_j, we have

$$\sum_{\tau_i \in \mathcal{D}_j \setminus \mathcal{D}_{j-1}} f_i = \sum_{\tau_i \in \mathcal{D}_{j-1} \setminus \mathcal{D}_j} f_i + f_j^{\mathcal{D}} - f_{j-1}^{\mathcal{D}}.$$

Since

$$\sum_{\tau_i \in \mathcal{D}_j} f_i = \sum_{\tau_i \in \mathcal{D}_j \setminus \mathcal{D}_{j-1}} f_i + \sum_{\tau_i \in \mathcal{D}_j \cap \mathcal{D}_{j-1}} f_i,$$

we have the flow conservation constraint holds at vertex v_j if and only if

$$\begin{aligned}\sum_{\tau_i \in \mathcal{D}_j} f_i &= \sum_{\tau_i \in \mathcal{D}_{j-1} \setminus \mathcal{D}_j} f_i + f_j^{\mathcal{D}} - f_{j-1}^{\mathcal{D}} + \sum_{\tau_i \in \mathcal{D}_j \cap \mathcal{D}_{j-1}} f_i \\ &= \sum_{\tau_i \in \mathcal{D}_{j-1}} f_i + f_j^{\mathcal{D}} - f_{j-1}^{\mathcal{D}}\end{aligned}$$

Finally, by the induction hypothesis, the above equation is equivalent to

$$\sum_{\tau_i \in \mathcal{D}_j} f_i = f_{j-1}^{\mathcal{D}} + f_j^{\mathcal{D}} - f_{j-1}^{\mathcal{D}} = f_j^{\mathcal{D}}$$

This complete the proof. □

Lemma 4.6: There exists a feasible flow in graph G if Inequalities 4.6, 4.7 are satisfied for interval int^k and furthermore the PO-set utilizations satisfy the following condition.

$$\forall \mathcal{D}_j \subset \mathcal{T} : u_j^{\mathcal{D}} \leq \frac{|\mathsf{int}^k| - 1}{|\mathsf{int}^k|} \tag{4.11}$$

Proof: First note that Inequalities 4.6 are necessary for the edge constraints on each PO-set edge (Inequality 4.8) to be satisfied. Let us construct a flow as follows.

$$\forall \tau_i \in \mathcal{T} : f_i = \mathsf{lag}(\tau_i, \mathsf{int}^k)$$
$$\forall \mathcal{D}_j \subset \mathcal{T} : f_j^{\mathcal{D}} = \mathsf{lag}(\mathcal{D}_j, \mathsf{int}^k)$$

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 4.9) and the left-side edge constrains of each PO-set edge (Inequality 4.8) 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 4.7, before the execution of GenerateLoad for interval int^k we have the following:

$$\begin{aligned}&\mathsf{lag}(\mathcal{D}_j, \mathsf{int}^k) = u_j^{\mathcal{D}} * t^{k+1} - \sum_{\tau_i \in \mathcal{D}_j} \sum_{x \in [0, t^k)} S(\tau_i, x) \\ &\leq u_j^{\mathcal{D}} * t^{k+1} - \lfloor u_j^{\mathcal{D}} * t^k \rfloor \\ &< u_j^{\mathcal{D}} * t^{k+1} - u_j^{\mathcal{D}} * t^k + 1.\end{aligned}$$

Now by Inequalities 4.11, the following holds:

$$\mathsf{lag}(\mathcal{D}_j, \mathsf{int}^k) < u_j^{\mathcal{D}} * t^{k+1} - u_j^{\mathcal{D}} * t^k + 1 \leq |\mathsf{int}^k|.$$

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 4.5 proves this statement. □

Lemma 4.7: If there is an integral feasible flow in graph G, then there is a feasible load set where $\forall \tau_i \in \mathcal{T} : l_i^k = f_i$.

Proof: Given an integral feasible flow, $\forall \tau_i \in \mathcal{T}$ let $l_i^k = f_i$. The following inequality holds.

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

Thus the interval loads satisfy Inequality 4.4. We now have to prove that the interval loads also satisfy Inequality 4.5. By the necessary condition of Lemma 4.5, we have $\sum_{\tau_i \in \mathcal{D}_j} f_i = f_j^{\mathcal{D}}$. Then since $\sum_{\tau_i \in \mathcal{D}_j} l_i^k = \sum_{\tau_i \in \mathcal{D}_j} f_i$ and $f_j^{\mathcal{D}}$ is subject to PO-set edge constraints in Inequality 4.8, Inequality 4.5 is satisfied. □

We can finally state our main theorem.

Theorem 4.2: acyclic transaction set $\mathcal{T}$ is schedulable by POGen if:

$$\forall \mathcal{D}_j \subset \mathcal{T} : u_j^{\mathcal{D}} \leq \frac{\mathsf{L} - 1}{\mathsf{L}}.$$

Proof: Since $\mathsf{L} \leq \min_k(|\mathsf{int}^k|)$, Inequalities 4.11 hold. Assume that Inequalities 4.6, 4.7 hold for a specific interval int^k. Then by Lemma 4.6 and [14], the constructed graph G has an integral feasible flow. Hence, by Lemma 4.7 algorithm GenerateLoad computes a feasible load set, which proves Proposition 4.1. Since furthermore, according to Lemma 4.3, Inequalities 4.6, 4.7 hold for every interval int^k, it follows that Inequalities 4.4 and 4.5, and therefore feasibility Conditions 1 and 2, also hold for every interval. This concludes the proof. □

Algorithm analysis: Since $\forall g_i \in E : c_i - b_i \leq 1$ and $\forall g_j^{\mathcal{D}} \in E : c_j^{\mathcal{D}} - b_j^{\mathcal{D}} \leq 1$, we have $\Delta = \sum_{g_i \in E} (c_i - b_i) + \sum_{g_j^{\mathcal{D}} \in E} (c_j^{\mathcal{D}} - b_j^{\mathcal{D}}) \leq N + N^{\mathcal{D}} \leq 2N$. 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 $\Delta \leq 2N$, the time complexity of GenerateLoad is $O(N^2)$. Finally, since the time complexity of POBase is $O(N*\max(\log(N), |\mathsf{int}^k|))$, the time complexity of POGen to generate the schedule for int^k is $O(N * \max(N, |\mathsf{int}^k|))$.

5 Scheduling Algorithms for Cyclic Transaction Sets

The cyclic transaction set scheduling problem is NP-complete because the special case where all transmission times are 1 and all periods are equal has been shown to be NP-complete [14][2]. In this section we will propose an approximation algorithm for this problem. The proposed solution uses the transaction buffer at a bus element to transform a cyclic transaction set into an acyclic one such that the latter's schedule can be used to execute the former. More specifically, we select a bus element ϵ_k and split each transaction τ_i that go through ϵ_k into two *pseudo transactions* τ_i' and τ_i''. τ_i' transfers data of τ_i from ϵ_i^1 to ϵ_k, and τ_i'' transfers the data which is stored in ϵ_k by τ_i' to ϵ_i^2. We said that τ_i' and τ_i'' is *feasibly transferred* data of τ_i if data of every job of τ_i is transferred to ϵ_i^2 before its deadline. The new transaction set is acyclic since there is no transaction going through ϵ_k. However, there is a precedence constraint between the pseudo transactions i.e. if τ_i'' transferred data in slot t then that data must be stored in ϵ_k by τ_i' before t. Since this constraint is not an assumption of transaction sets that can be scheduled by POGen, τ_i' and τ_i'' may not feasibly transfer data of τ_i. This happens when τ_i'' has the same transmission time as τ_i and is scheduled when there is no data of τ_i stored in ϵ_k. We deal with this problem by increasing the transmission time of τ_i' and τ_i'' to be more than that of τ_i. However the increment in the transmission time of pseudo transactions causes their utilization to be bigger than that of τ_i. The question is how to minimize this increase. We will address this question in Lemma 5.2 after we formally describe the problem in the next paragraphs.

As described above, we replace each transaction $\tau_i \in \mathcal{T}$ where $\tau_i = (e_i, p_i, \epsilon_i^1, \epsilon_i^2)$ and τ_i goes through ϵ_k with two *pseudo transactions* (p-transactions) τ_i' and τ_i'' where $\tau_i' = (e_i^+, p_i, \epsilon_i^1, \epsilon_k)$, $\tau_i'' = (e_i^+, p_i, \epsilon_k, \epsilon_i^2)$, and $e_i^+ > e_i$. τ_i' and τ_i'' have the same utilization $u_i^+ = e_i^+/p_i > u_i$. τ_i is called the *original transaction* (o-transaction) of τ_i' and τ_i''. The new transaction set is called *pseudo transaction set* denoted by $\mathcal{T}'$. The following (work conserving) rule is applied to the schedules of a p-transaction.

Rule 1: A p-transaction *always* transfer data of the current job of its o-transaction in slot t if there is data of the job stored in the source element of the p-transaction at time t.

2. The scheduling problem of the special case is equivalent to the Circular-Arc Coloring Problem [14].

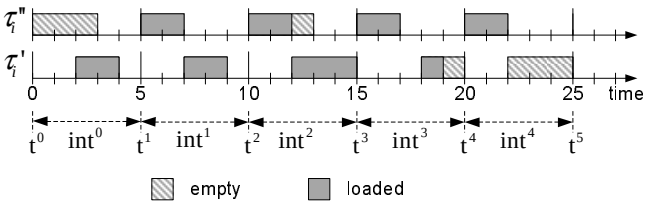

Fig. 8. Schedule of p-transactions where L =5(slots), $\forall k : \mathsf{int}^k = \mathsf{L}$, $\tau_i = (8, 5\mathsf{L}, \epsilon_i^1, \epsilon_i^2)$, and $\tau_i' = (12, 5\mathsf{L}, \epsilon_i^1, \epsilon_k)$, $\tau_i'' = (12, 5\mathsf{L}, \epsilon_k, \epsilon_i^2)$

Note that the execution of a p-transaction in slot t can transfer data of the current job of its o-transaction only if the data has already been stored in its source elements before time t, otherwise the execution does nothing. In the former case, we say that the execution slot of the p-transaction is *loaded*, and is *empty* in the latter. Note that τ_i' execution slots are always loaded until it transfers all the data of the current job of τ_i. It is because when a job of τ_i' is ready, a job of τ_i is also ready and all data of the job of τ_i has been stored in ϵ_i^1. Therefore, the statement is true by Rule 1. Figure 8 shows an example of the schedule of τ_i' and τ_i'' with the given transaction parameters and with every scheduling interval having size L. Consider int^0 in which τ_i' has 2 execution slots and τ_i'' has 3 execution slots (this number is determined by function GenerateLoad). Since τ_i' is scheduled in slot 2 and 3 (this schedule is determined by POBase), there is no data of τ_i stored in ϵ_k before time $t = 3$. Therefore, the 3 execution slots of τ_i'' in int^0 are empty.

We say that a p-transaction is *effective in execution slot t* when either of the following cases occurs: 1) the execution slot is loaded or 2) the execution slot is empty and the p-transaction has transferred *all* data of the current job of its o-transaction at time t. Note that τ_i' is always effective in all slots because its execution slots are always loaded until it transfers all the data of the current job of τ_i. However that is not the case for τ_i''. In Figure 8, τ_i'' is not effective in slot 0, 1, 2, and 12. In these slots, there is still data of the current job of τ_i stored at ϵ_i^1 but there is no data of the job stored at ϵ_k. The following lemma is obvious due to Rule 1.

Lemma 5.1: Consider job j of τ_i which is ready at t and has deadline at $t + p_i$, and consider scheduling interval int^k where $t \leq t^k < t^{k+1} \leq t + p$. If τ_i'' is not effective in one of its execution slot in int^k, then, at t^{k+1}, the amount of data of j that has been transferred by τ_i'' must be at least equivalent to $\lfloor u_i^+ * t^k \rfloor - u_i^+ * t$ slots.

Proof: By Rule 1, if τ_i'' is not effective in int^k, then by t^{k+1}, τ_i'' must have transferred all the data of j which had been transferred by τ_i' until t^k and there must be a portion of data of j still being stored at ϵ_i^1 after t^k. This also means that all execution slots of τ_i' between t and t^k are loaded. By POGen, the number of execution slots of τ_i' between t (i.e. when job j is ready) and t^k are $\lfloor u_i^+ * t^k \rfloor - u_i^+ * t$. Since τ_i' is always loaded in these slots, the amount of data of j that has been transferred

by τ_i' until t^k is equivalent to at least $\lfloor u_i^+ * t^k \rfloor - u_i^+ * t$ slots. This complete the proof. □

We will now describe how to determine e_i^+ such that τ_i is schedulable and u_i^+ is minimized.

Lemma 5.2: Every job of τ_i completes before its deadline if $\mathcal{T}'$ is schedulable by POGen and

$$\forall \mathsf{int}^k : e_i^+ \geq \left\lfloor \frac{e_i + 1}{\alpha} \right\rfloor, \tag{5.1}$$

where $\alpha = 1 - |\mathsf{int}^k|/p_i$.

Proof: Since $\mathcal{T}'$ is schedulable by POGen, jobs of τ_i' and τ_i'' complete before their deadlines which are the same as the deadline of the correspondent job of τ_i. Thus, to complete the proof, we only need to show that the jobs of τ_i' and τ_i'' together transfer all the data of τ_i during their execution.

Consider job j of τ_i which is ready at t and has deadline at $t + p_i$. Let int^k where $t \leq t^k < t^{k+1} \leq t + p_i$ be the last scheduling interval where τ_i'' is not effective in at least one slot in int^k. We have:

- By POGen, the maximum number of execution slots of τ_i'' within $[t, t^{k+1})$ is $\lceil u_i^+ * t^{k+1} \rceil - u_i^+ * t$. Therefore, the smallest number of execution slots of τ_i'' within $[t^{k+1}, t + p_i)$ is $A = e_i^+ - \lceil u_i^+ * t^{k+1} \rceil + u_i^+ * t$.
- By Lemma 5.1, The amount of data of j that τ_i'' needs to transfer after t^{k+1} is equivalent to at most $B = e_i - \lfloor u_i^+ * t^k \rfloor + u_i^+ * t$ slots.

Since τ_i'' is always effective within $[t^{k+1}, t + p_i)$, all data of τ_i will be transferred to ϵ_i^2 before the deadline if

$$\begin{aligned} & A \geq B \Leftrightarrow \\ & e_i^+ - \lceil u_i^+ * t^{k+1} \rceil + u_i^+ * t \geq e_i - \lfloor u_i^+ * t^k \rfloor + u_i^+ * t \\ & \Leftrightarrow e_i^+ - e_i \geq \lceil u_i^+ * t^{k+1} \rceil - \lfloor u_i^+ * t^k \rfloor. \end{aligned} \tag{5.2}$$

Since

$$\begin{aligned} \lceil u_i^+ * t^{k+1} \rceil - \lfloor u_i^+ * t^k \rfloor &\leq \lceil u_i^+ * t^{k+1} \rceil - \lceil u_i^+ * t^k \rceil + 1 \\ &\leq \lceil u_i^+ * |\mathsf{int}^k| \rceil + 1, \end{aligned}$$

if the following inequality is true then Inequality 5.2 is also true:

$$e_i^+ - e_i \geq \lceil u_i^+ * |\mathsf{int}^k| \rceil + 1 \tag{5.3}$$

Since e_i^+ is integer, Inequality 5.3 is equivalent to

$$\lceil e_i^+ * \alpha \rceil \geq e_i + 1. \tag{5.4}$$

We complete the proof by showing in the following that if e_i^+ satisfies 5.1 then 5.4 is true:

$$\begin{aligned} \lceil e_i^+ * \alpha \rceil &\geq \left\lceil \left\lfloor \frac{e_i + 1}{\alpha} \right\rfloor * \alpha \right\rceil \geq \left\lceil \left(\frac{e_i + 1}{\alpha} - 1 \right) * \alpha \right\rceil \\ &= \lceil (e_i + 1) - \alpha \rceil = e_i + 1. \end{aligned}$$

The last equation in the above derivation comes from the fact that e_i is integer and $\alpha < 1$. □

If we set every scheduling interval such that its length is the GCD of all periods, i.e. $\forall k : |\mathsf{int}^k| = \mathsf{L}$, then u_i^+ is minimized when

$$e_i^+ = \left\lfloor \frac{e_i + 1}{1 - \mathsf{L}/p_i} \right\rfloor. \tag{5.5}$$

The scheduling algorithm for cyclic transaction set $\mathcal{T}$, namely cPOGen, is summarized as follows.

Algorithm cPOGen:

- Step 1: Find ϵ_k such that the derived pseudo transaction set $\mathcal{T}'$, whose execution time of the p-transactions are calculated by Equation 5.5, passes the sufficient utilization bound test of POGen.
- Step 2: Schedule $\mathcal{T}'$ using POGen in which every scheduling interval has its length to be L, and schedule $\mathcal{T}$ accordingly following Rule 1.

Algorithm analysis: Step 1 of algorithm cPOGen only needs to be executed once and can be implemented with time complexity $O(N^2)$. Step 2 is a POGen algorithm therefore it has the same time complexity as POGen.

6 Evaluation

Most of the previous related works [22], [23], [4], [16], [19] 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 [22] a branch-and-bound algorithm that searches for a feasible priority set for a transaction set. If a feasible priority set exists, then the transactions set is guaranteed to be schedulable under the worst-case transaction latency (WTL) analysis proposed in [23]. The works in [23], [22] are the state of the art.

In this section, we are interested in comparing the performance of POGen/cPOGen on MDRB with the solution proposed in [23], [22]. The analyzed performance metric is the percentage of random transaction sets which are schedulable under POGen/cPOGen and FPA. Under POGen/cPOGen, an acyclic transaction set is schedulable if it passes the utilization bound test of Theorem 4.2, whereas a cyclic one is schedulable if its pseudo transaction set is schedulable. A transaction set is schedulable under FPA if it has a feasible priority set generated by the algorithm in [22].

In our experiments, we used three controlled parameters: 1) the maximum PO-set utilization of a transaction set[3]; 2) the size of transaction sets; and 3) the number of bus elements. The transactions' sources and destinations are randomly selected from the set of bus elements. Meanwhile, the transactions' utilization, transmission time and period are generated as follows. Given a maximum PO-set utilization u^{max}, the utilization of transaction τ_i is initially generated according to the uniform distribution algorithm in [6] such that the utilizations of all PO-sets are no larger than u^{max}. The transmission time e_i is generated as a uniformly-distributed random number in the range of 1 to 100 slots. The period p_i is then determined as $\lceil e_i/(u_i * \mathsf{L}) \rceil * \mathsf{L}$. Finally, given the pair of $\{e_i, p_i\}$, we recalculate u_i to be e_i/p_i.

3. it is equivalent to "the maximum link utilization" in [22]

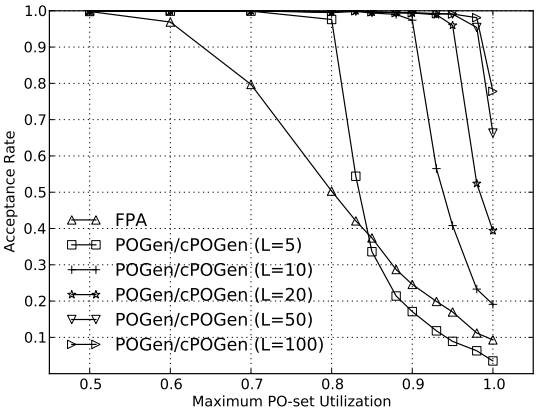

Fig. 9. Acceptance rate with various maximum PO-set utilization

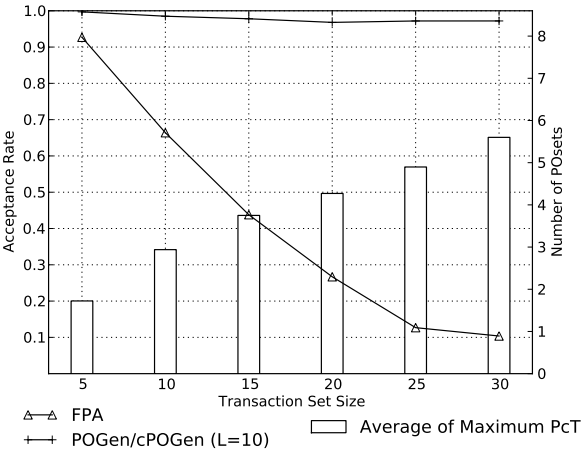

Fig. 10. Acceptance rate with various size of transaction sets

The following graphs depict the average acceptance rate of POGen/cPOGen and FPA over 1000 different random transaction sets, in each of which two controlled variables are kept constant while the other one is varied. The set of transaction sets used to draw each graph has both acyclic and cyclic transaction sets.

Figure 9 shows the average acceptance rate of FPA and POGen/cPOGen (when L is 5, 10, 20, 50, or 100) under various maximum PO-set utilization. In these experiments, the size of transaction sets and the number of bus elements are set at 20 and 10, respectively. In this setting, approximately 80% of the generated transaction sets are cyclic. It can be seen that in most cases the acceptance rate of POGen/cPOGen is better than that of FPA especially when PO-set maximum utilization is high and $L > 5$. The better performance of POGen/cPOGen comes from the fact that the WTL analysis in [23] does not always take advantage of the parallelism between non-overlapping transactions. For example, consider the transaction set shown in Figure 1. Assume τ_3 and τ_5 have higher priority than τ_4. According to the WTL analysis in [23], the interference of transactions τ_3 and τ_5 on the execution of τ_4 is calculated as if all transactions were using a single-shared resource. However, POGen/cPOGen allows τ_3 and τ_5 to be executed in parallel as shown in Figure 4. In other words, the acceptance rate of FPA will be reduced when the number of distinct PO-sets that contain a given transaction increases. Let denote this number as PcT.

Figure 10 shows the acceptance rate with the maximum PO-set utilization to be 0.9, the number of bus elements to be 10, and various size of transaction sets. We also draw in Figure 10 a bar graph which shows the average (over all transaction sets) of the maximum PcT in each set. The performance of POGen/cPOGen is better than FPA especially when the size of transaction sets are higher. The reason is that, when the number of bus elements is fixed, the higher the size of a transaction set, the bigger the maximum PcT (as shown in the bar graph). As a consequence, FPA suffers more from the effect described in the previous paragraph. The performance of POGen/cPOGen, however, stays almost the same.

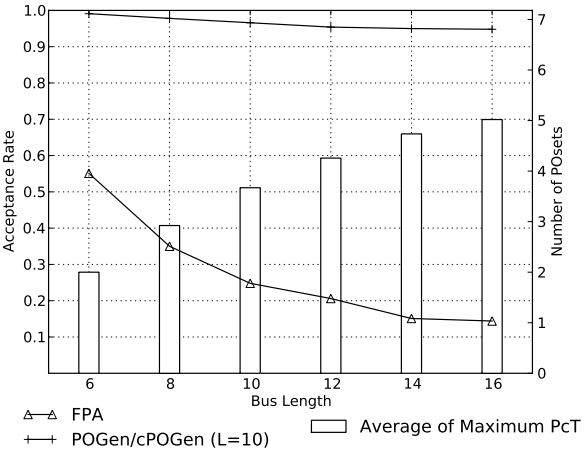

Fig. 11. Acceptance rate with various bus length

The same reason explains the better performance of POGen/cPOGen in Figure 11 which shows the acceptance rate of POGen/cPOGen (with $L = 10$) and FPA when the number of bus elements is varied. In these experiments, the maximum PO-set utilization is set at 0.9, the size of transaction sets are fixed at 20. When the number of bus elements increases, there will be more longer transactions which may belong to higher number of distinct PO-sets as shown with the bar graph.

7 Implementation

In this section, we discuss a POGen implementation on a Cell Broadband Engine processor [9] and report its execution time overhead. The Cell processor has 1 PowerPC Processing Element (PPE) and 8 Synergy Processing Elements (SPE) each of which is an element on the Cell ring bus. There are also 3 additional bus elements which are a memory controller and two I/O controllers.

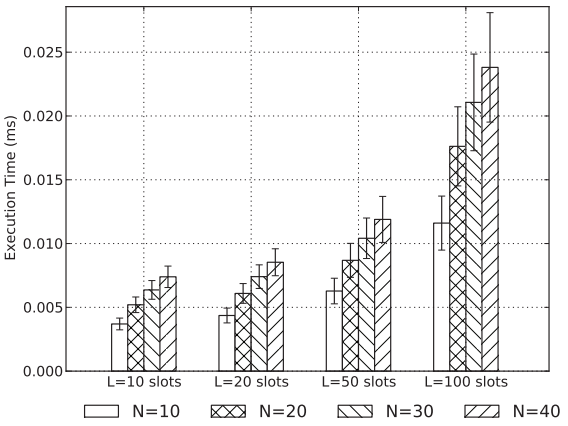

Fig. 12. Average execution time of POGen

POGen is implemented to run on SPEs as an online algorithm. It is invoked at the beginning of each scheduling interval by a timer-interrupt handler and generates the schedule of all transactions in that interval. Then, if the generated schedule has $S(\tau_i, t) = 1$, a slot scheduler will transfer data of τ_i in slot t using Direct-Memory-Access commands. The slot scheduler is also invoked by a timer-interrupt handler. The execution of POGen and the slot scheduler induce some overheads. We describe the measurement of these overheads in the next paragraphs.

POGen execution time was measured under various slot sizes which are $10us$, $20us$, $50us$, and $100us$. We assume that the period of every transaction is a multiple of $1ms$ which is also the smallest possible scheduling interval. We also selected the size of every scheduling interval to be equal to the size of L which happened to be $1ms$ in all generated transaction sets. Since L is $1ms$, given the different values of slot size, the size of L measured in number of slots are 100, 50, 20 and 10 slots. We generated transaction sets with various sizes using the same methodology discussed in Section 6. The sizes of transaction sets are 10, 20, 30, and 40.

Figure 12 shows the execution time of POGen in ms under various conditions. This execution time also includes the latency of the timer-interrupt handler that invokes POGen. It can be seen that the algorithm overhead increases when L has a higher value. However the algorithm overhead is no more than $0.03ms$ even when $\mathsf{L} = 100$ slots. Since each scheduling interval is $1ms$, under the given conditions, the maximum algorithm overhead is less than 3% of the scheduling interval size.

Our measurement also shows that the execution time of the slot scheduler is no more than $0.125us$. In other words, if the slot size is $10us$, the overhead is less than 1.25% of the slot size. The overhead is smaller when the slot size is bigger.

8 Conclusion

We have investigated the problem of real-time communication scheduling on multi-core processor buses with ring topology. This scheduling problem has important assumptions that are different with traditional real-time problems. We proposed a novel scheduling algorithm to solve the problem at hand. Compared to previous works, our algorithms employ a dynamic-priority scheduling scheme and can achieve much higher bus utilization. Our future works will focus on extending the proposed algorithms to other bus topology such as mesh and torus.

References

[1] Cell be programming tutorial. IBM, 2007.

[2] Ankur Agarwal, Cyril Iskander, and Ravi Shankar. Survey of network on chip (noc) architectures & contributions. *Journal of Engineering, Computing and Architecture*, 3(1), 2009.

[3] T. W. Ainsworth and T. M. Pinkston. Characterizing the cell eib on-chip network. *IEEE Micro*, 27(5):6–14, 2007.

[4] Shobana Balakrishnan and Füsun Özgüner. A priority-driven flow control mechanism for real-time traffic in multiprocessor networks. *IEEE Trans. Parallel Distrib. Syst.*, 9(7):664–678, 1998.

[5] 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.

[6] Enrico Bini and Giorgio C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Syst.*, 30(1-2), 2005.

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

[8] Bach D. Bui, Rodolfo Pellizzoni, Deepti K. Chivukula, and Marco Caccamo. Real-time communication for multicore systems with multi-domain ring buses. In *RTCSA '10: Proceedings of the 16th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, Macau, China.

[9] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: A performance view. IBM Research, 2005.

[10] Jason Howard et al. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *IEEE International Solid-State Circuits Conference*, 2010.

[11] 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.

[12] Sathish Gopalakrishnan, Lui Sha, and Marco 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.

[13] Philip Holman and James H. Anderson. Adapting pfair scheduling for symmetric multiprocessors. *J. Embedded Comput.*, 1(4):543–564, 2005.

[14] Jon Kleinberg and Éva Tardos. *Algorithm Design*. Addison Wesley, March 2005.

[15] John P. Lehoczky and Lui Sha. Performance of real-time bus scheduling algorithms. *SIGMETRICS Perform. Eval. Rev.*, 14(1):44–53, 1986.

[16] Jong-Pyng Li and Matt W. Mutka. Real-time virtual channel flow control. *J. Parallel Distrib. Comput.*, 32(1):49–65, 1996.

[17] C. L. Liu and James W. 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] Zhonghai Lu, Axel Jantsch, and Ingo Sander. 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*, pages 960–964, New York, NY, USA, 2005. ACM.

[20] Marco Di Natale and Antonio Meschi. Scheduling messages with earliest deadline techniques. *Real-Time Syst.*, 20(3):255–285, 2001.

[21] Lionel M. Ni and Philip K. Mckinley. A survey of wormhole routing techniques in direct networks. *IEEE Computer*, 26:62–76, 1993.

[22] Zheng Shi and Alan Burns. Priority assignment for real-time wormhole communication in on-chip networks. In *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium*, pages 421–430, Washington, DC, USA, 2008. IEEE Computer Society.

[23] Zheng Shi and Alan 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.

[24] K. Tindell, A. Burns, and A. Wellings. Analysis of hard real-time communications. *Real-Time Systems*, 9:147–171, 1995.

[25] Dakai Zhu, Daniel Mosse, and Rami Melhem. Multiple-resource periodic scheduling problem: How much fairness is necessary. In *24th IEEE International Real-Time Systems Symposium*, 2003.