Real-time Communication for Multicore Systems with Multi-domain Ring Buses

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

Abstract

We address the problem of scheduling real-time data transactions on a multicore processor bus. In particular, to increase system predictability and tighten WCET estimation, we propose to employ a software-controllable Multi-Domain Ring Bus (MDRB) architecture. The problem of scheduling periodic real-time transactions on MDRB is challenging because the bus allows multiple non-overlapping transactions to be executed concurrently, and because the degree of concurrency depends on the topology of the bus and of executed transactions. We propose a practical abstraction mechanism for the scheduling problem together with two novel scheduling algorithms. 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.

1 Introduction

A hard real-time system has a strict requirement that deadlines of all tasks must be guaranteed therefore their worst case execution times (WCET) must be reliably estimated. Unfortunately, modern computer architectures are typically designed in such a way that the estimation of a tight bound for WCET is very difficult. A significant source of unpredictability lies in the interconnection architecture used by the CPU and Direct-Memory-Access (DMA)-enabled peripherals to communicate among each other and with main memory. Most commercial-off-the-shelf architectures employ shared buses such as the Front Side Bus (FSB) between CPU and main memory and the PCI bus for peripherals. These shared buses can easily become an *uncontrollable bottleneck* in the system. The authors of [19] observed that, due to contention for access to the shared infrastructure between a CPU and a DMA-enabled sensor peripheral, a task execution time may be unexpectedly extended up to 44%. The problem is even more severe in multicore systems as more entities can simultaneously compete for access to the bus; the authors of [20] measured an extension in task execution time of 196% in a dual-core system.

There have recently been significant research efforts on interconnection architectures for multicore processors which have features suited for real-time systems, especially in the field of networks-on-chip. These works are thoroughly surveyed in [2]. Commercial multicore processors with software-controlled interconnections have also been developed. For example, the IBM Cell Broadband Engine Architecture (CellBE) [6] is well-distinguished for its high performance. The CellBE consists of nine processing elements. All of them connect to each others and to main memory and I/O devices through a software-controlled ring bus.

Such software-controlled Multicore Processor Buses (MPB) can significantly help to increase system predictability and tighten WCET estimation. Consider highly critical real-time systems such as avionics and medical systems. A typical task in such systems executes the following activities: 1) it collects data from sensors that are tracking some physical events; 2) it processes the data; 3) it sends processed data or control signals to other tasks or actuators. An example can be a multipurpose status display task on an avionic system [15] 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 is the thread streaming model [1] in which different real-time tasks run on different processing elements. Data transactions between tasks, I/O devices and main memory are executed through the MPB. The key idea is that since MPB accesses are software-controlled, software designers can schedule these data transactions deterministically.

The main research issues are how to provide software designers with: 1) a practical and accurate abstraction of the real-time scheduling problem on MPB; 2) an effective scheduling methodology that optimizes MPB utilization. In this paper, we address these problems on a specific MPB architecture, the Multi-Domain Ring Bus (MDRB) architecture. In a MDRB, transactions that do not overlap (i.e., they are not routed through the same bus segment) can be transferred concurrently. MDRB has been implemented in commercial systems [6, 3] and is a cost-effective high-performance solution for MPB. Real-time coordination of data transactions while maximizing MDRB utilization is challenging because of two reasons: 1) a MDRB can carry multiple transactions concurrently; 2) transactions delay each other if they overlap. At first glance, scheduling transactions on the bus 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 MPB depends on the *topology* of the bus and executed transactions.

This research has three main contributions. First, we propose

a scheduling abstraction that can provide an accurate view of the real-time performance of a MDRB, while abstracting away the details of the low-level physical bus implementation. Therefore, the abstraction and its scheduling algorithms can be used for most MDRB implementations. In contrast, many previous works on real-time scheduling for networks-on-chip have focused on specific low-level implementation such as the network using wormhole routers [22, 21, 13, 4]. Second, we propose two scheduling algorithms for a common subset of transaction topologies (non-circular transactions): POBase and POGen. POBase is optimal under restrictive assumptions on transactions' timing parameters. POGen induces a sufficient schedulable utilization bound for general transaction sets, which we show to be highly competitive for typical MDRB implementations. To the best of our knowledge, our algorithms are the first *dynamic-priority* algorithms proposed for MPB; most previous works [22, 21, 13, 4] have focused on fixed-priority scheduling. Third, we implemented a bus scheduling engine on a CellBE platform and performed experiments to demonstrate the effectiveness of real-time enforcement on bus transactions.

The paper is organized as follows. We briefly provide background knowledge on the Multi-Domain Ring Bus in Section 2 and survey related works in Section 3. Section 4 defines the real-time bus transactions and the scheduling model. In Section 5, we discuss our proposed real-time scheduling algorithms for MDRB. Section 6 discusses an implementation of the bus scheduling engine in a real system together with experimental results. We conclude our paper in Section 7.

2 Background

The performance of a Multicore Processor Bus (MPB) is usually measured based on three metrics: concurrency, scalability, and cost and energy efficiency. The Multi-Domain Ring Bus (MDRB) architecture has been shown to have a good balance between these metrics [3] based on commercial implementations [6]. The ring architecture requires smaller amount of wires compared to other common network-on-chip architectures such as torus and mesh. As a consequence, this architecture is simpler, supports higher clock rates and is a cost effective approach for the interconnection of multiple processing elements. Unlike the traditional single-domain bus architecture where all elements contend for a single transmission medium, MDRB achieves higher concurrency by allowing transactions between elements to be transferred in parallel if they do not use the same physical bus wires, i.e. a bus segment between any two elements.

An example of a commercial multi-core processor employing a MDRB is the IBM Cell Boardband Engine (CellBE) [6]. All elements of CellBE connect through a ring bus. Data transactions between elements can be transferred concurrently on the ring if they do not use the same bus segment. All transactions between any two of the eight CellBE Synergistic Processing Elements (SPE) and between system memory and a SPE are explicitly initiated by software running on the SPE.

In this paper, we are concerned with an interconnection architecture similar to that of CellBE. More specifically, we assume that the bus has a ring architecture in which each bus element has direct connections to only two neighboring elements. A *data* transaction is defined as a request made by an application to transfer a certain amount of data between two bus elements. Each element has a bus router that is able to transmit a transaction toward its destination. 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 [18] 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 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. However, multiple non-overlapping data transactions can be sent at the same time.

We are interested in solving the real-time scheduling problem for a MDRB with one ring. In particular, we focus on bus systems with one bidirectional ring since they are most common; however, the proposed scheduling algorithms can also be applied to bus systems with two unidirectional rings (one clockwise and one counterclockwise) by analyzing each direction separately. We plan to study buses with more complex architectures as part of our future work.

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, in our model, 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. As we show in Section 6, the abstraction typically introduces little overhead because atomic transactions belonging to the same unit transactions are pipelined on the bus.

3 Related Works

Many of the early works on hard real-time communication [12, 23, 17] 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 [9]. 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 [14] is applied or extended to solve the problem in these works. The MPB systems 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 MPB. Most of these works [21, 22, 4, 13, 16] are concerned with the *fixed-priority* scheduling paradigm. For example, in [21, 22], 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. Since the works on fixed-priority scheduling analyze the schedulability in terms of the WTL while that of our work is a utilization bound, a direct comparison between approaches is not immediately obvious and will be left as future work. However, we believe that the performance of our approach is competitive as it has been the case in microprocessor real-time scheduling.

4 Real-time Bus Transaction and Scheduling Model

We consider a scheduling problem where applications request periodic data transfers (data transactions) on the bus. A periodic data transaction comprises an infinite sequence of jobs.

Without loss of generality, let the bus elements be indexed clock-wise. 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, and $\epsilon_i^1, \epsilon_i^2$ are the two indexes of the two endpoints which are called the first and the second index of τ_i, respectively. Each job must complete within its period, i.e. relative deadlines are equal to periods. A transaction has two endpoints ϵ_i^1 and ϵ_i^2 if it uses all bus elements in $[\epsilon_i^1, \epsilon_i^2]$ in the clockwise direction. τ_i is said to *go through* element ϵ if $\epsilon \in [\epsilon_i^1, \epsilon_i^2)$ (excluding the second endpoint 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 transfered concurrently on the bus if they use the same bus segment between any two elements. Based on the endpoint definition, it is obvious that two transactions overlaps if and only if they go through the same element. 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.

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 non-overlapping transaction 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 $N^{\mathcal{D}}$. Notice that although τ_2 and τ_6 have one same endpoint (element 5), they do not overlap because they do not share any bus segment. Since each PO-set contains at least one element different from those of other PO-sets and transactions are arranged in an one dimensional space, $N^{\mathcal{D}} \leq N$.

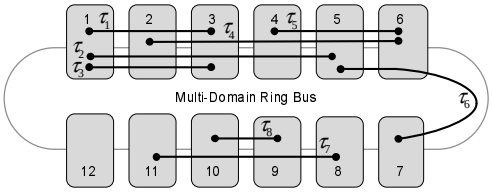

Figure 1. Non-circular transaction set

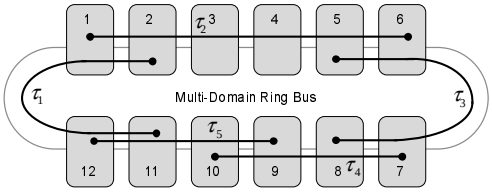

Figure 2. Circular transaction set

A transaction set is said to be *circular* if its overlapping transactions create a cycle on the bus and to be *non-circular* otherwise. Figure 1 and Figure 2 show an example of a non-circular and a circular transaction set, respectively. A non-circular transaction set can be represented as a set of overlapping intervals on an indexed straight line where each interval corresponds to a transaction and the straight line is indexed by the indexes of the bus elements. Figure 3 shows the indexed straight line representation of the non-circular transaction set shown in Figure 1. Let the left-most transaction on the straight line be the *first* transaction. If there are more than one left-most transactions, any one of them can be the first transaction. For simplicity, we index the bus elements such that the first transaction has the first index to be the smallest index.

Due to the discrete nature of transactions' execution times and periods, we adopt the discrete scheduling model used in [5]. More specifically, we assume that every transaction's execution time and period are integral values. Scheduling decision is also made at integral values of time, 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. 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

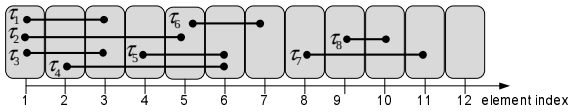

Figure 3. Indexed straight line representation

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 4.1.

Theorem 4.1 *A transaction set $\mathcal{T}$ is schedulable only if:*

$$\forall \mathcal{D} \subset \mathcal{T} : u^{\mathcal{D}} = \sum_{\forall \tau_i \in \mathcal{D}} u_i \leq 1 \tag{4.1}$$

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, Inequlity 4.1 must be satisfied. □

Let $\mathcal{E}(\epsilon_k)$ be a set of all transactions in $\mathcal{T}$ that go through same bus element ϵ_k. The following lemma is necessary for later discussion.

Lemma 4.1 *Given a transaction set $\mathcal{T}$ that satisfies the necessary condition, the following inequality holds.*

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

Proof.
Since transactions in $\mathcal{E}(\epsilon_k)$ pairwise overlap, there exists $\mathcal{D}$ such that $\mathcal{E}(\epsilon_k) \subseteq \mathcal{D}$. Therefore the lemma is implied by Theorem 4.1. □

5 Scheduling Algorithms for Ring Buses

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

First, we propose an algorithm, namely POBase, which schedule every non-circular transaction set whose transactions have the *same* period. We will prove that the necessary condition (Theorem 4.1) is also the sufficient condition for same-period non-circular 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 non-circular transaction sets whose transactions do not have the same period. POGen, which is built based on POBase, can schedule all transaction sets whose PO-set utilizations satisfy the following utilization bound:

$$\forall \mathcal{D} \subset \mathcal{T} : u^{\mathcal{D}} \leq \frac{\mathsf{L}-1}{\mathsf{L}}, \tag{5.1}$$

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 [15]. As we will show in the implementation section, with the speed of the state of the art multicore chip buses [3], the practical time slot size is about $1us$ to $10us$. Meanwhile, the period granularity in practical real-time applications [15] is at the level of milliseconds. That means L has practical values ranging from 100 to 1000 time units.

Finally, we will discuss the issue of scheduling circular transaction sets and our proposed initial solution.

5.1 The POBase algorithm

The problem of scheduling a non-circular same-period transaction set is similar to the problem of interval graph vertex coloring [8]. However, the optimal coloring algorithm in [8] can only schedule transactions which all have the same execution time. POBase is a modification of this algorithm to handle the problem at hand. POBase is a first-fit algorithm with respect to a transaction ordering. More specifically, in POBase, the transactions are ordered by their first index. Then in ascending order, each transaction is assigned to the earliest slots where no smaller-ordered overlapping transaction has been already assigned to[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 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 $\{2, 5, 6, 7\}$.

Algorithm 1 POBase

Input: transaction set $\mathcal{T}$ such that $\forall \tau_i \in \mathcal{T} : p_i = p$ where p is a constant
Output: schedule S for period p

1: $\mathcal{L} \leftarrow$ the list of all $\tau_i \in \mathcal{T}$ ordered according to ϵ_i^1
2: **for** each $\tau_i \in \mathcal{L}$ in ascending order **do**
3: **for** each $t \in [0, p)$ **do**
4: **if** $\sum_{x \in [0,p)} S(\tau_i, x) < e_i$ **then**
5: **if** $\forall \tau_j \in \mathcal{L} : OV(\tau_i, \tau_j) = 0$ or $S(\tau_j, t) = 0$ **then**
6: $S(\tau_i, t) \leftarrow 1$
7: **end if**
8: **end if**
9: **end for**
10: **end for**

Theorem 5.1 POBase *is optimal for same-period non-circular transaction sets.*

Proof.
The generated schedule is valid because the condition at Step 5 guarantees that a transaction is not scheduled in the same slot with its overlapping transactions. It remains to show that if transaction set satisfies the necessary condition, then at the end of the algorithm,

$$\forall \tau_i : \sum_{x \in [0,p)} S(\tau_i, x) = e_i. \tag{5.2}$$

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

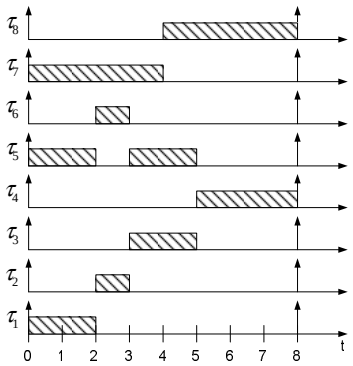

Figure 4. An example of the POBase algorithm

We will prove this by induction.
Base case: Consider the first iteration of the for-loop starting at Step 2. In this iteration, the schedule of τ_1 in $\mathcal{L}$ is generated. Since τ_1 is the first transaction whose schedule is generated and $e_1 \leq p$, at the end of the iteration, Equation 5.2 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 2, Equation 5.2 holds for all transactions $\{\tau_i : i \in [1,k]\}$. We will prove that Equation 5.2 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 $\mathcal{E}(\epsilon^1_{k+1})$ be the set of transactions that go through ϵ^1_{k+1}. By the way the transactions are ordered and since $\mathcal{T}$ is non-circular, we have that $\forall \tau_i \in \mathcal{T}$ if $OV(\tau_i, \tau_{k+1}) = 1$ and $\sum_{x\in[0,p)} S(\tau_i, x) > 0$ then $\tau_i \in \mathcal{E}(\epsilon^1_{k+1})$. In other words, among all the transactions that overlap with τ_{k+1}, only transactions in $\mathcal{E}(\epsilon^1_{k+1})$ have their schedule been generated. Therefore, the contradiction assumption occurs only when:

$$\sum_{\tau_i\in\mathcal{E}(\epsilon^1_{k+1})} \sum_{x\in[0,p)} S(\tau_i, x) = p. \tag{5.3}$$

Since the following is true:

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

by the contradiction assumption and Equation 5.3 we have: $\sum_{\tau_i\in\mathcal{E}(\epsilon^1_{k+1})} e_i > p$. This contradicts with Lemma 4.1 which implies that $\sum_{\tau_i\in\mathcal{E}(\epsilon^1_{k+1})} e_i \leq p$. Therefore, at the end of the iteration, Equation 5.2 must hold for τ_{k+1}. This completes the proof. □

Algorithm analysis: An efficient sorting algorithm has time complexity $O(N)$. In addition, Step 5 can be implemented to have a time complexity of $O(N)$. Therefore the time complexity of POBase to build a schedule of p slots for N transactions is $O(N^2 * p)$.

5.2 The POGen algorithm

In this subsection we propose a scheduling algorithm (POGen) for non-circular 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^k_i 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 [10]. 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:

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

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 [7, 24]. However, since none of these works uses the transaction overlap assumption, their proposed algorithms can not be used for the problem at hand. In POGen, a *scheduling interval* is defined as the interval between two closest arrival times (also deadlines) of any two transactions. Figure 5 shows an example of the scheduling intervals induced by the set of three transactions $\tau_1 = \{e_1 = 1, p_1 = 2, \epsilon^1_1 = 1, \epsilon^2_1 = 3\}$, $\tau_2 = \{e_2 = 1, p_2 = 3, \epsilon^1_2 = 1, \epsilon^2_2 = 4\}$ and $\tau_3 = \{e_3 = 1, p_3 = 6, \epsilon^1_3 = 2, \epsilon^2_3 = 5\}$.

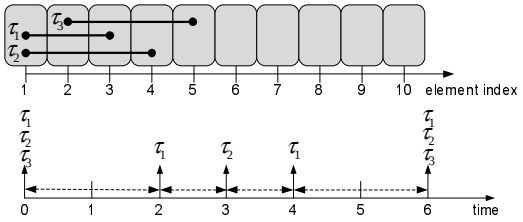

Figure 5. Scheduling intervals on the execution time line

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 [10]. 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:

$$\forall \tau_i \in \mathcal{T} : \lfloor \mathsf{lag}(\tau_i, \mathsf{int}^k) \rfloor \leq l_i^k \leq \lceil \mathsf{lag}(\tau_i, \mathsf{int}^k) \rceil, \tag{5.4}$$

$$\forall \mathcal{D} \subset \mathcal{T} : \lfloor \mathsf{lag}(\mathcal{D}, \mathsf{int}^k) \rfloor \leq \sum_{\tau_i \in \mathcal{D}} l_i^k \leq |\mathsf{int}^k|. \tag{5.5}$$

Inequality 5.4 sets conditions on the interval load for each transaction, based on the closest integral values of the lag functions. Inequality 5.5 sets conditions on the total interval load of each PO-set. Note that the right side of Inequality 5.5 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 5.4, 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 5.5 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. 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. In Lemma 5.1, we first show that the following inequalities initially hold for int^0:

$$\forall \mathcal{D} \subset \mathcal{T} : \lfloor \mathsf{lag}(\mathcal{D}, \mathsf{int}^k) \rfloor \leq |\mathsf{int}^k|, \tag{5.6}$$

$$\forall \mathcal{D} \subset \mathcal{T} : \sum_{\tau_i \in \mathcal{D}} \sum_{x \in [0, t^k)} S(\tau_i, x) \geq \lfloor u^{\mathcal{D}} * t^k \rfloor. \tag{5.7}$$

A feasible load set is then computed in Step 2 of POGen by the GenerateLoad procedure, which is assumed to honor the following proposition:

Proposition 5.1 *Assume that all PO-sets satisfy the utilization bound in Inequalities 5.1. If Inequalities 5.6, 5.7 hold for* int^k*, then* GenerateLoad *computes a feasible load set for* int^k.

Given a feasible load set for interval int^0, Lemma 5.1 guarantees that Inequalities 5.6, 5.7 again hold for int^1. Hence, GenerateLoad can be used to 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 5.3, we will prove that GenerateLoad indeed honors Proposition 5.1.

Algorithm 2 POGen

Input: transaction set $\mathcal{T}$
Output: schedule S

1: **for** each interval int^k ordered by increasing k **do**
2: $\{l_i^k : \forall i \in [1, N]\} \leftarrow$ GenerateLoad($\mathcal{T}$,int^k)
3: $\mathcal{T}' \leftarrow \{\{l_i^k, |\mathsf{int}^k|, \epsilon_i^1, \epsilon_i^2\} : \forall i \in [1, N]\}$
4: S for interval $\mathsf{int}^k \leftarrow$ POBase($\mathcal{T}'$)
5: **end for**

Lemma 5.1 *If* GenerateLoad *honors Proposition 5.1, then Inequalities 5.6, 5.7 hold for every scheduling intervals.*

Proof.
We prove by induction.
Base step: Consider the first scheduling interval $\mathsf{int}^0 = [0, t^1)$. Inequalities 5.6 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 5.7 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 5.6, 5.7 hold in every scheduling interval up to int^k. We prove that Inequalities 5.6, 5.7 also hold before the execution of GenerateLoad at interval int^{k+1}. Since Inequalities 5.6, 5.7 are satisfied at interval int^k, GenerateLoad generates a feasible load set and POBase generates a feasible schedule for the interval. Therefore after Step 4, 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 5.5, we obtain the following which proves that Inequalities 5.7 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 5.6. Notice that since $S(\tau_i, x)$ is integer, we have:

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

Since Inequalities 5.7 hold for int^{k+1}, Inequalities 5.6 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. □

5.3 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 [11]. 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 5.1, there always exists a feasible solution therefore answering Question 1. Then, since the Ford-Fulkerson algorithm [11] 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 $\mathcal{D}_j$. For each vertex, a *PO-set edge* $g_j^{\mathcal{D}}$ is defined which exits from the vertex and whose flow value $f_j^{\mathcal{D}}$ represents the interval load of the corresponding PO-set. A lower bound value $b_j^{\mathcal{D}}$ and a capacity $c_j^{\mathcal{D}}$ are defined for each of the PO-set edges such that Inequalities 5.5 are imposed on their flow values:

$$\forall \mathcal{D}_j \subset \mathcal{T} : b_j^{\mathcal{D}} = \lfloor \mathsf{lag}(\mathcal{D}_j, \mathsf{int}^k) \rfloor \leq f_j^{\mathcal{D}} \leq c_j^{\mathcal{D}} = |\mathsf{int}^k|. \quad (5.8)$$

Furthermore, for each transaction τ_i, a *transaction edge* is defined whose flow value 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 5.4 are imposed on their flow values:

$$\forall \tau_i \in \mathcal{T} : b_i = \lfloor \mathsf{lag}(\tau_i, \mathsf{int}^k) \rfloor \leq f_i \leq c_i = \lceil \mathsf{lag}(\tau_i, \mathsf{int}^k) \rceil. \quad (5.9)$$

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

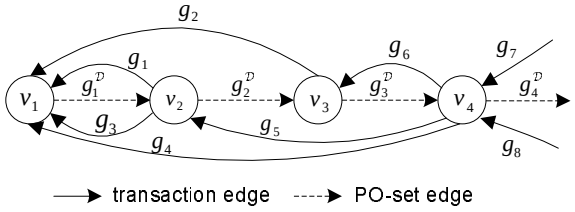

Figure 6. Constructed graph G

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 6 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 $c_j^{\mathcal{D}} = |\mathsf{int}^k|$ 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 5.2 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 [11] 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 $\mathcal{D}_2$. 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 5.2 shows that G is indeed a directed graph.

Lemma 5.2 *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 indexes. 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 5.1 and therefore POGen generates feasible schedules for all transaction sets that satisfy the utilization bound of Inequalities 5.1. For simplicity of exposition, we split the proof in multiple lemmas. The following Lemma 5.3 shows that a feasible load set can be found from an integral feasible flow of the correspondent graph G. Then, Lemma 5.5 proves that graph G has a feasible flow if Inequalities 5.6, 5.7 are satisfied for interval int^k and furthermore all PO-sets satisfy an utilization constraint based on $|\mathsf{int}^k|$. Note that we know from [11] that if graph G has a feasible flow, then it has an integral feasible flow which can be found by the Ford-Fulkerson algorithm [11]. Finally, we will show that the utilization bound of Inequalities 5.1 implies the utilization bound used in Lemma 5.5. Hence, GenerateLoad honors Proposition 5.1.

Lemma 5.3 *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 5.4. We now have to prove that the interval loads also satisfy Inequality 5.5. We prove this by induction over the ordered set of vertices.
Base case: by the flow conservation constraint at vertex v_1, we have

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

Then, by the edge constraints of PO-set edge $g_1^{\mathcal{D}}$, the Inequalities 5.5 of $\mathcal{D}_1$ hold.

$$\lfloor \mathsf{lag}(\mathcal{D}_1, \mathsf{int}^k) \rfloor \leq \sum_{\tau_i \in \mathcal{D}_1} l_i^k \leq |\mathsf{int}^k|$$

Induction case: Assume Inequality 5.5 is satisfied up to $\mathcal{D}_{j-1}$ and the following induction hypothesis holds.

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

We prove that Inequality 5.5 and the induction hypothesis is also satisfied for $\mathcal{D}_j$. Given the induction assumption and the flow conservation constraint at vertex v_j, the following equalities hold.

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

Then, by the edge constraints of PO-set edge $g_j^{\mathcal{D}}$, Inequality 5.5 holds for $\mathcal{D}_j$.

$$\lfloor \mathsf{lag}(\mathcal{D}_j, \mathsf{int}^k) \rfloor \leq \sum_{\tau_i \in \mathcal{D}_j} l_i^k \leq |\mathsf{int}^k|$$

Furthermore, the induction hypothesis also holds.

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

This complete the proof. □

The following lemma is necessary to prove Lemma 5.5.

Lemma 5.4 *The following equalities hold for every vertex v_j*

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

Proof.
The lemma is a direct result from the induction hypothesis in the proof of Lemma 5.3. □

Lemma 5.5 *There exists a feasible flow in graph G if Inequalities 5.6, 5.7 are satisfied for interval int^k and furthermore the PO-set utilizations satisfy the following condition.*

$$\forall \mathcal{D}_j \subset \mathcal{T} : u_j^{\mathcal{D}} \leq \frac{|\mathsf{int}^k| - 1}{|\mathsf{int}^k|} \tag{5.10}$$

Proof.
First note that Inequalities 5.6 are necessary for the edge constraints on each PO-set edge (Inequality 5.8) to be satisfied. Let us construct a flow as follows.

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

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 5.9) and the left-side edge constrains of each PO-set edge (Inequality 5.8) 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 5.7, 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 5.10, 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 constrains at each vertex also hold. By Lemma 5.4, the total flow value entering vertex v_j can be calculated as:

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

which equals to the total flow value exiting v_j i.e.:

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

□

We can finally state our main theorem.

Theorem 5.2 *Non-circular 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 5.10 hold. Assume that Inequalities 5.6, 5.7 hold for a specific interval int^k. Then by Lemma 5.5 and [11], the constructed graph G has an integral feasible flow. Hence, by Lemma 5.3 algorithm GenerateLoad computes a feasible load set, which proves Proposition 5.1. Since furthermore, according to Lemma 5.1, Inequalities 5.6, 5.7 hold for every interval int^k, it follows that Inequalities 5.4 and 5.5, and therefore feasibility Conditions 1 and 2, also hold for every interval. This concludes the proof. □

Algorithm analysis: The time complexity of the Ford-Fulkerson algorithm is also the time complexity of GenerateLoad and is equal to $O(N * |\mathsf{int}^k|)$. Therefore, the time complexity of POGen is $O(N^2 * h)$.

5.4 Scheduling circular transaction sets

Unfortunately, neither algorithm POBase nor POGen can be applied to circular transaction sets. According to Theorem 5.1, the condition of Theorem 4.1 is both necessary and sufficient for same period non-circular transactions; however, it is not sufficient for circular transactions. As an example, consider the transaction set in Figure 2. Assume that all transactions have transmission times equal to 1 and periods equal to 2. The utilization of each PO-set of this transaction set is 1, hence it satisfies the necessary condition. However, it is easy to see that there does not exist any feasible schedule. In fact, any valid schedule can have at most four transactions scheduled in the first two slots, therefore the fifth transaction misses its first deadline.

We plan to study the scheduling of circular transactions in more details as part of our future work. We would like to stress that in most cases, engineering approaches can be used to replace a circular transaction set with a functionally equivalent non-circular one. We believe that in many applications, software designers can avoid the transaction cycle by selecting the endpoints of transactions. That is possible because endpoints are determined by the placement of software tasks which produce or use the transaction data. When this technique can not be applied, we propose a solution to convert a circular transaction set into a non-circular one as follows: select an element ϵ on the ring; then for each transaction τ_i that goes through ϵ, split τ into two transactions such that one of them has endpoints $\{\epsilon_i^1, \epsilon\}$ and the other one has endpoints $\{\epsilon, \epsilon_i^2\}$. Note that the two transactions do not overlap, hence the cycle is broken at ϵ. However, to respect the ordering between transactions it might be necessary to buffer the transaction data at ϵ for one full transaction period.

6 Implementation

In this section, we discuss the implementation of a bus scheduling engine on a real system. We also show the effect of the scheduling enforcement on the real-time behavior of bus transactions. More specifically, we show that real-time transactions of feasible transaction sets can miss the deadlines when executed without enforcement, while complete before deadline when executed by the schedule generated by Algorithm POGen.

Scheduling engine implementation: We have chosen Sony Play Station 3 (PS3) as the experimental platform. PS3 uses a CellBE processor [6] whose architecture closely matches the bus model that we are interested in (see Section 2). We implemented a scheduling engine whose instances run on each processing element. The engine is in fact a timer interrupt handler. When a timer interrupt fires at the beginning of each time slot, the scheduling engine will make a scheduling decision for the current slot. The scheduling decision is made based on a scheduling table which is generated off-line by the POGen algorithm or by executing the algorithm itself. We chose table-based approach in our following experiments. If a transaction is scheduled in a slot, a DMA package of the transaction with a given size will be transfered in the slot. The size of a DMA package is dictated by the bus bandwidth and the slot size.

Experimental results: Although the CellBE bus has two rings in each direction, transaction layouts exist that use only one ring in each direction. We have chosen one of those lay-

DMA size (KB)	8	10	12	14	16
Transmission time (ticks)	34	41	48	54	60

Table 1. DMA transmission time

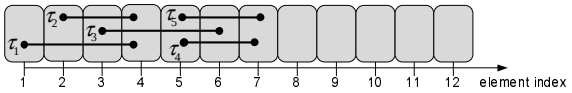

Figure 7. Experimental transaction set

outs for our experiments as our algorithm assumes the bus has one ring in each direction. In addition, all transactions in the experiments have endpoints that are SPEs.

In order to determine the time slot size and the amount of data to transfer in each slot, we measured the transmission time of DMA packages with various sizes. The results are shown in Table 1 where a tick is a SPE real-time clock tick and is equal to $12.5ns$. All transmission times are rounded up to the smallest integral number of ticks. It can be seen that the bus achieves a higher bandwidth with a bigger DMA package size. Our measurement also shows that the time a SPE spends to execute a timer interrupt handler to transfer a DMA package is 2 ticks. In our later experiments, we choose a slot size to be 43 ticks and DMA package size in each slot to be $10KB$. The slot size includes 41 ticks of a 10KB DMA package transmission time and 2 ticks of the interrupt handler processing overhead.

In order to show the effect of the real-time scheduling enforcement, we perform an experiment with a transaction set with five transactions whose layout is shown in Figure 7 and with values (unit in number of slots) of timing parameters equal to $\tau_1 : e_1 = 4, p_1 = 20; \tau_2 : e_2 = 6, p_2 = 10; \tau_3 : e_3 = 6, p_3 = 60; \tau_4 : e_4 = 6, p_4 = 10; \tau_5 : e_5 = 4, p_5 = 20$. The transaction set has $\mathsf{L} = 10$ and two PO-sets: $\mathcal{D}_1 = \{\tau_1, \tau_2, \tau_3\}$ and $\mathcal{D}_2 = \{\tau_3, \tau_4, \tau_5\}$. The utilization of each PO-set is $9/10 = (\mathsf{L} - 1)/\mathsf{L}$. In the first experiment, we initiate each transaction as soon as it arrives in the system, hence the transactions are scheduled by the low-level round-robin bus arbiter. As the result, transaction τ_2 and τ_4 *miss* their deadline with the maximum relative completion time of 12 slots as opposed to their relative deadline of 10 slots. When the transaction set is scheduled by POGen, all transactions *meet* their deadlines.

7 Conclusion

We have investigated the problem of real-time communication on multicore processor buses with ring topology and proposed two novel scheduling algorithms for real-time bus transactions. Compared to previous work, our algorithms employ a dynamic-priority scheduling scheme and can achieve high bus utilization. Our future works will focus on: 1) addressing the issue of transforming circular transaction sets into non-circular ones; 2) 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] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell broadband engine architecture and its first implementation: A performance view. IBM Research, 2005.

[7] Hyeonjoong Cho, Binoy Ravindran, and E. Douglas Jensen. An optimal real-time scheduling algorithm for multiprocessors. *Real-Time Systems Symposium, IEEE International*, 0:101–110, 2006.

[8] Martin Charles Golumbic. *Algorithmic Graph Theory and Perfect Graphs (Annals of Discrete Mathematics, Vol 57)*. North-Holland Publishing Co., Amsterdam, The Netherlands, The Netherlands, 2004.

[9] 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.

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

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

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

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

[14] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[15] C. D. Locke, D. R. Vogel, L. Lucas, and J. B. Goodenough. Generic avionics software specification. Technical Report CMU/SEI-90-TR-8, 1990.

[16] 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.

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

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

[19] R. Pellizzoni, B. D. Bui, M. Caccamo, and L. Sha. Coscheduling of cpu and i/o transactions in cots-based embedded systems. In *Proceedings of the 29th IEEE Real-Time Systems Symposium*, 2008.

[20] R. Pellizzoni, A. Schranzhofer, J.-J. Chen, M. Caccamo, and L. Thiele. Worst case delay analysis for memory interference in multicore systems. In *Proceedings of Design, Automation and Test in Europe (DATE)*, Dresden, Germany, March 2010. Accepted for publication. Available at http://netfiles.uiuc.edu/rpelliz2/www/.

[21] 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.

[22] 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.

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

[24] 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.