Scheduling Real-Time Constant Bit Rate Flows over a TDMA Channel

Nir Naaman and Raphael Rom

Department of Electrical Engineering
Technion - Israel Institute of Technology
Haifa 32000, Israel
{mnir@tx, rom@ee}.technion.ac.il

December 17, 2002

Abstract

We consider a scheduling problem in which real-time Constant Bit Rate (CBR) flows must be scheduled over a TDMA channel. Scheduling is performed by a central scheduler which is responsible for all bandwidth allocations. Each flow has QoS requirements that include bit rate, delay, and delay jitter. In order to provide the requested QoS a flow must get fixed size bandwidth allocations at periodic intervals. Our model of the problem is derived from a scheduling problem that appears in centralized access networks such as CATV, broadband wireless access, and passive optical networks. In these networks real-time CBR flows are used to deliver voice and other real-time applications that generate fixed size data packets on a periodic basis.

The scheduling problem is analyzed in both its offline and online settings. We focus on the case where grant intervals are an integer multiple of each other. In this case the scheduling problem can be modelled as a variant of bin packing where bin sizes can be modified in a constrained manner. We show that deciding whether a set of CBR flows can be scheduled is NP-complete whenever there are two or more different grant intervals. Several scheduling algorithms are suggested and their performance is investigated. We relate the performance of the algorithms to parameters such as grant sizes and tolerated jitter, and derive conditions under which our scheduling algorithms are optimal.

Contents

List of Figures

1 Introduction

We examine a scheduling problem which is present in several networks with a Time Division Multiple Access (TDMA) Medium Access Control (MAC) protocol. In particular we refer to the class of centralized broadcast networks, examples of which are data over cable networks, passive optical networks, and Broadband Wireless Access networks such 802.16 (fixed wireless MAN), satellite and 3G cellular networks. In centralized broadcast networks a central station, called the Headend or hub, is connected to all other station in the network by a broadcast channel. The central station is responsible for allocating bandwidth to all other stations in the network. It allocates the available bandwidth by assigning time slots to different stations. Each station is allowed to transmit only in the time slots that have been assigned to it by the central station. The scheduling problem is that of performing the bandwidth allocation to the different stations in the best possible way.

This paper considers the problem of allocating bandwidth to flows with Quality of Service (QoS) requirements. In particular we examine the task of scheduling Constant Bit Rate (CBR) real-time flows. A CBR flow has both bandwidth and timing requirements; it must get periodic bandwidth allocations so as to ensure delivering the guaranteed Quality of Service. Each flow is characterized by three parameters: grant size, grant interval, and grant jitter. The flow must get a grant (an allocation) equal to its grant size every grant interval time units. The grant jitter specifies the tolerated delay of the actual grant from the nominal grant time (more details in Section 2). CBR flows are used by many applications for the transmission of voice and non compressed video.

We study both offline and online versions of the scheduling problem. In the offline version the entire set of flows is known to the scheduler before the schedule begins, while in the online version flows are established over time and each new flow must either be scheduled or rejected. In both cases the goal is normally to maximize the channel utilization, or to schedule as many flows as possible. The general scheduling problem, where flows may have arbitrary grant sizes and intervals, is complex. We therefore analyze the problem in stages by considering increasingly more complex cases. In Section 4 we study the case of a single grant interval. In this case the answer to most of the scheduling problems we present is easy. In Section 5 we consider the case where there are two different grant intervals, assuming these intervals are an integer multiple of each other. We show that in this case the problem of deciding whether a given set of flows has a legal schedule is NP-complete. We analyze the scheduling problem by converting it to a variant of bin packing in which bin sizes can be modified in a constrained manner. The complexity of the problem is analyzed as a function of the tolerated jitter and conditions that ensure a legal schedule exists for a given set of flows are derived. In Section 6 we present several approximation algorithm for the problem and investigate their performance. Section 7 extends the analysis to cases where there are flows with more than two related grant intervals, while Section 8 addresses the problem when grant intervals are not related. In Section 9 we study the online scheduling problem.

As we mentioned, the scheduling problem we study is present in many communication networks. Our model of the problem and the terminology we use are derived from data over cable networks. In particular we refer to Data-Over-Cable Service Interface Specification (DOCSIS) [1]. The standard is published by CableLabs [2] and is the leading standard for data over cable networks. In the next subsection we present an overview of scheduling CBR flows as defined by the DOCSIS standard. We omit many details which are not directly relevant to our scheduling problem, see [1] for the complete description.

1.1 Scheduling CBR Flows in Data over Cable Networks

Data over cable has emerged as one of the leading technologies for delivering broadband services over the local loop. The existing infrastructure of Hybrid-Fiber-Coax (HFC), used by most modern CATV networks, enables cable operators to provide bi-directional high bandwidth data channels, at relatively low cost. Initially cable operators offered only best effort type of service by sharing the available bandwidth equally among all subscribers. In recent years, however, cable operators have been looking for ways to provide additional services such as telephony and video. Since these new services cannot rely on best effort type of service, a mechanism for supporting different Quality of Service requirements has been devised. In this paper we consider the mechanism used for supporting flow with Constant Bit Rate type of service. The main application of CBR flows is the delivery of voice over the CATV media. Cable operators intend to use CBR flows to offer Voice over IP (VoIP) services that would compete with telephony services offered by telephone companies (see e.g., [3], [4] and [5] for more details).

CATV networks are characterized by a tree-and-branch topology. The Headend or Cable Modem Termination System (CMTS), at the root of the tree, controls all traffic in the network. Subscribers of data services use a Cable Modem (CM) to connect to the CMTS. The available bandwidth is divided into channels. Downstream channels (CMTS to CMs) are used only by the CMTS. Upstream channels (CMs to CMTS) are shared by many subscribers (typically 500 to 2000). To share an upstream channel a TDMA MAC protocol with dynamic bandwidth allocation is implemented. As the downstream is used only by the Headend no MAC protocol is needed.

DOCSIS based networks transfer Internet Protocol (IP) datagrams between the CMTS and the CMs. The CMTS is responsible for the scheduling of all transmissions in the upstream. Scheduling is done by dividing the upstream, in time, into a sequence of numbered mini-slots. A mini-slot is the unit of granularity for upstream transmission; transmitting a datagram may require one or more mini-slots. The CMTS and a CM establish a *service flow* between them; a service flow describes the type of connection between the CMTS and the CM and is identified by a service identifier (SID). A CM can support multiple active service flows simultaneously. In order to support different QoS demands DOCSIS defines several types of service flows. The main services are Unsolicited Grant Service (UGS) intended for CBR flows, Real-Time Polling Service (rtPS) intended for Variable Bit Rate (VBR) flows, Non-Real-Time Polling Service (nrtPS) intended for non real-time VBR flows, and Best Effort service. We are interested in the Unsolicited Grant Service which is designed to support real-time service flows that generate fixed size data packets on a periodic basis. Today, the main application of UGS flows is the delivery of VoIP packet telephony calls. When a user wishes to make a new VoIP call the CM tries to establish a UGS service flow with the CMTS. The CMTS decides whether to accept or reject the call; if the call is accepted the CMTS must allocate a fixed number of mini-slots to the CM at periodic intervals (see [5] for exact description). An active UGS service flow is characterize by three parameters: grant size (number of bytes to be allocated in each grant), grant interval (the delay between successive grants), and a tolerated grant jitter. Each flow may have a different set of parameters depending on factors such as the type of CODEC (Coder-Decoder) used, the bandwidth and buffering requirements of the application, and the number of active sessions (grants per interval) supported by the flow. The scheduling algorithm at the CMTS must allocate the available mini-slots to the different service flows while ensuring that each service flow is scheduled according to its specifications.

2 Problem Definition and Notations

We consider the problem of scheduling a set $F = \{f_1, \ldots, f_n\}$ of CBR flows over a slotted TDMA channel. A slot is the basic time unit in the system. Each flow $f_i \in F$ is characterized by three parameters

- Grant Size - $S(i)$ - the number of slots that must be allocated to the flow in each grant.
- Grant Interval - $I(i)$ - the nominal time between every two grants to f_i. The Grant Interval is expressed in units of slots.
- Grant Jitter - $J(i)$ - also referred to as the *tolerated jitter*, this value defines the tolerated delay from the nominal grant time until the actual grant time. The Grant Jitter is expressed in units of slots.

Scheduling Rules: Each flow f_i is associated with a time reference which we denote by $t_1(i)$. The time reference is assigned to the flow by the scheduling algorithm and defines the nominal grant times of the flow; the k^{th} nominal grant time of flow f_i is $t_k(i) = t_1(i) + (k-1)\,I(i)$. Flow f_i must get an allocation of $S(i)$ consecutive slots in each grant. The actual grant time must not precede the nominal grant time and must not exceed it by more than the grant jitter. To state it formally, let $\tau_k(i)$ be the actual time of the k^{th} grant of f_i, then it must satisfy $t_k(i) \leq \tau_k(i) \leq t_k(i) + J(i)$.

We define the *actual jitter* of the k^{th} grant of f_i as $j_k(i) \equiv \tau_k(i) - t_k(i)$, where $t_k(i)$ and $\tau_k(i)$ are the nominal and actual times of the k^{th} grant of flow f_i, respectively. To maintain a legal schedule of f_i the actual jitter of each grant must not exceed the grant jitter, i.e., $0 \leq j_k(i) \leq J(i)$, $\forall k$.

Our objective is to schedule the flows such that the timing requirements of all the flows are maintained. We call a schedule that achieves this goal a ***legal schedule***. We call a set of flows for which a legal schedule exists a ***feasible set***. Figure 1 presents an example of a legal schedule.

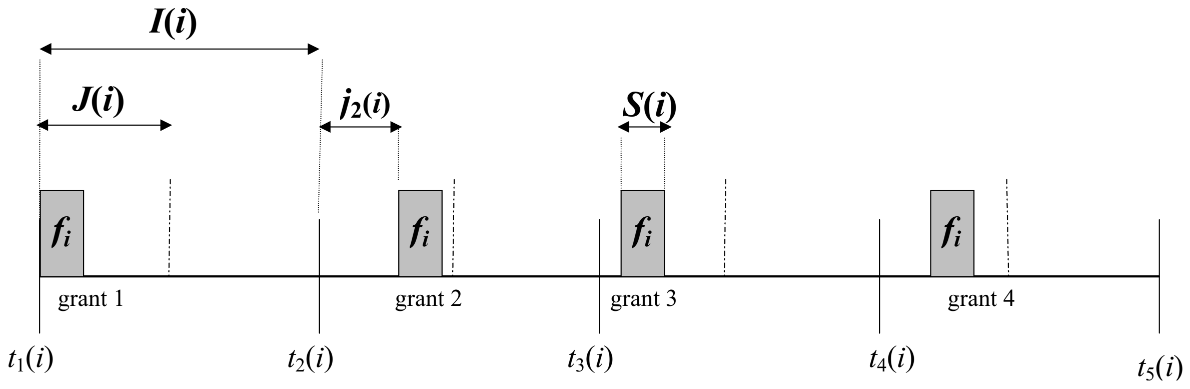

Figure 1: Example of a legal schedule of flow f_i.

The scheduling problem appears in several variations

- Offline setting - Before scheduling begins the entire set of flows is already known to the scheduling algorithm and all flows are ready to be scheduled.
- Online setting - Flows are established over time; scheduling decisions are made without any knowledge of future flows.
- Permanent flows - Flows with infinite duration, i.e., a flow that has been scheduled lasts forever.

- Temporary flows - Each flow has a duration for which it must be scheduled. Once a flow completes its duration it is removed. In the online problem the duration may be known or unknown when the flow arrives.

We are interested in the following problems:

1. **Feasible-Set** - Given a set of flows F, is the set feasible, i.e., is there a legal schedule for the set?
2. **Legal-Schedule** - Given a feasible set of flows, find a legal schedule for the set.
3. **Optimal-Subset** - Given a set of flows F, find an optimal feasible subset $F' \subseteq F$. We consider two optimization function: (1) maximize the number of flows in F' and (2) maximize the channel utilization, that is, maximize $W_{F'} \equiv \sum_{f_i \in F'} S(i)/I(i)$.
4. **Admission Control** - Given a legal schedule of a set of flows, can we add an additional flow f without violating the scheduling rules?

2.1 Notations

For a given set F of n flows with K different values of grant intervals we denote by $I_1 \leq I_2 \leq \ldots \leq I_K$ the different grant intervals sorted in increasing order. We say that the grant intervals are *related* if they are integer multiple of each other, that is, $I_j = m_j \cdot I_{j-1}$ where m_j is an integer and $2 \leq j \leq K$. We define K disjoint groups $F_1, \ldots, F_K$ such that $F_j = \{f_i \in F : I(i) = I_j\}$ where $1 \leq j \leq K$, that is, group F_j contains all flows in F with grant interval equal to I_j.

For each group F_j we define the following attributes:

- S_j is the maximal grant size of flows in F_j.
- J_j is the minimal grant jitter of flows in F_j.
- $W_{F_j} \equiv \sum_{f_i \in F_j} S(i)/I(i)$ is the total bandwidth requirements of flows in F_j.

Similarly, for the entire set F we define

- $S_{max} = \max\{S_j\}$ is the maximal grant size of all flows in F.
- $J_{min} = \min\{J_j\}$ is the minimal grant jitter of all flows in F.
- $W_F \equiv \sum_{f_i \in F} S(i)/I(i)$ is the total bandwidth requirements of all flows in F.

2.2 Performance Evaluation of Algorithms

To evaluate the performance of the different algorithms we study we use competitive analysis. In this type of analysis the performance of an algorithm is compared to the performance of an optimal algorithm. For a given algorithm A and a set of input flows F, we denote by $A(F)$ the cost (according to the given metric) of algorithm A when it is applied to the set F. We denote by $OPT(F)$ the cost of an optimal algorithm. In order to provide a uniform definition for both maximization and minimization problems we adopt a common notation [6] and define

$$R_A(F) \equiv \max\left\{\frac{A(F)}{OPT(F)}, \frac{OPT(F)}{A(F)}\right\} \tag{1}$$

We define the *worst case performance ratio* of algorithm A, R_A, as

$$R_A \equiv \inf\{r \geq 1 : \; R_A(F) \leq r \text{ for all sets } F\} \tag{2}$$

We define the *asymptotic worst case performance ratio* R_A^∞ as

$$R_A^\infty \equiv \inf\{r \geq 1 : \text{ for some } N > 0, \; R_A(F) \leq r \text{ for all } F \text{ with } \; OPT(F) \geq N\} \tag{3}$$

3 Related Work and Our Results

Our work is related to several fields of research that at first may seem different from one another. The difference is due to the fact that each field considers different applications and hence defines the scheduling problem in a different way. Our scheduling problem is very similar to problems that have been studied in the context real-time scheduling of periodic tasks. Research in this field has typically focused on single or multi processor scheduling and its applications are mainly from operational research and control systems. We cover related work in this field in subsection 3.1. Another related field is that of perfectly-periodic schedules. Our scheduling problem actually reduces to finding a perfectly-periodic schedule when the tolerated jitter of each flow is zero (see subsection 3.2). Finally in subsection 3.3 we mention related work that has been done on scheduling flows with QoS requirements in data networks and cable networks in particular.

3.1 Real-Time Scheduling of Periodic Tasks

There is a very reach literature on scheduling periodic real-time tasks (see e.g., surveys in [7], [8] and [9]). In this framework a set of n periodic tasks is to be scheduled on a single processor or multiple processors. Each task τ_i is characterized by the 4-tuple (e_i, p_i, d_i, r_i) where $e_i > 0$ is the task's execution time, $p_i > 0$ is the task's period, $d_i > 0$ is the task's deadline, and $r_i \geq 0$ is the task's initial release time. Task τ_i has a sequence of release times defined by $r_{i,k+1} = r_i + k\,p_i$, where $k \in Z^+$. The task must be scheduled periodically such that for every k the task is scheduled for e_i time units in the interval $r_{i,k} \leq t \leq r_{i,k} + d_i$. In some real time systems, in addition to timing constraints, there are also other constraints such as precedence or exclusion relations among tasks (see e.g, [10], [11]).

The relation between the above parameters and the parameters in our scheduling problem is as follows: $e_i = S(i)$, $p_i = I(i)$, $d_i = J(i)$, and $r_i = t_1(i)$. It is important to note that in our problem it is up to the scheduling algorithm to determine the time reference of each flow $(t_1(i))$. This means that our problem corresponds to scheduling problems where release times are not specified; such problems are harder than those where release times are specified. When we consider scheduling flows on a single channel our work corresponds to the single processor case, while scheduling in a multi channels network corresponds to the case of multi processors. Note that in our scheduling rules we assume fragmentation is not allowed and hence each grant must be over consecutive slots. This feature relates our work to non-preemptive scheduling. If we allow fragmentation a preemptive model (that takes fragmentation overhead into account) should be adopted.

Most of the algorithms that have been proposed for real time scheduling are priority based algorithms. In such algorithms each task τ_i is assigned a priority number by the algorithm. Priorities may be either static (constant over time) or dynamic (change over time). Lower priority numbers correspond to higher priorities. The algorithm selects to schedule the task with the highest priority among all available tasks (i.e., tasks that have been released); ties are broken arbitrarily. We list below four of the best known algorithms that have been considered for real time scheduling.

- Earliest Deadline First (EDF) [12] - This is a dynamic priority algorithm. A task is assigned a priority number equal to the time left before its deadline, that is, the priority number of a task τ_i at time t is equal to $d_i(t) - t$. As a consequence, the algorithm always schedules the task with the nearest deadline. EDF is one of the most significant scheduling algorithms in the field and has been the subject of many studies. The main reason is that EDF achieves good performance and is actually optimal in many cases (for example when preemption is allowed).

- Least Laxity First (LLF) [13] - A dynamic priority algorithm in which the priority of task τ_i at time t is equal to $d_i(t) - t - e_i(t)$. Thus LLF is similar to EDF but it also takes into account the execution time of the tasks.
- Rate Monotonic (RM) [14] - This is a simple static-priority scheduling algorithm in which priorities are equal to the periods of the associated tasks. Hence, the RM rule is to selects to schedule the (available) task with the shortest period.
- Deadline Monotonic (DM) [15] - This is also a static priority algorithm. DM uses the deadline span of each task for its priority. Thus the next task to schedule is the task with the smallest deadline span.

Most studies of real time scheduling assume that tasks can be preempted. Under this assumption Liu and Layland showed that the EDF algorithm is optimal in the sense that it produces a legal schedule for any schedulable set [12]. The subject of non-preemptive real time scheduling received less attention. Jeffay et al. considered non-preemptive scheduling on a single processor when the deadline of each task is equal to its period [16]. They presented necessary and sufficient conditions for a set of periodic or sporadic tasks to be schedulable for all values of release times. Furthermore, they showed that when these conditions are met the non-preemptive EDF algorithms is optimal. However, they proved that deciding whether a set of periodic tasks with specific release times is schedulable, is strongly NP hard. Improvements to the non-preemptive EDF algorithm for specified release times (where EDF is not optimal) have since been proposed (see e.g., [17], [18] and a survey in [19]). Several special cases of non-preemptive real-time scheduling have been studied. For equal size tasks it has been shown in [20] that when periods and release times are an integer multiple of the task size the non-preemptive EDF algorithm can schedule any set of tasks with bandwidth requirement that does not exceed unity. Dolev and Keizelman [21] studied the performance of the EDF and LLF algorithms for several cases of online scheduling of non periodic real time tasks. They derived several results that relate the performance ratio of the algorithms to the minimum and maximum task size.

The common assumption in most studies of real time scheduling has been that the deadline of a task is equal to its period (in our terminology $J(i) = I(i)$). The problem of general deadlines received less attention. Most of the works that have been published consider the problem of jitter control in scheduling periodic preemptive tasks (see e.g., [22], [23]). For non-preemptive tasks Di-Natale and Stankovic considered the case of scheduling periodic tasks with precedence and exclusion constraints [24]. They presented a general method, based on simulation annealing, of producing feasible schedules while minimizing the total jitter of all tasks. Another simulation annealing algorithm for scheduling tasks with jitter constraints (in addition to communication, precedence, and exclusion constraints) has been presented in [25]. We consider simulation annealing, as well as exhaustive search techniques, inappropriate for our scheduling problem due to their running time complexity.

3.2 Perfectly-Periodic Schedules

The common framework of perfectly-periodic schedules is that of n clients that share a single resource by means of time multiplexing. Each client i is characterized by its period P_i, and its request size R_i. A schedule is called perfectly-periodic if each client i is scheduled for R_i time slots exactly every P_i time slots. A perfectly-periodic schedule can be described in a very compact way; for each client it is sufficient to describe only an offset and a period. This fact renders perfectly-periodic schedules desirable in several applications. For example, in the context of broadcast disks [26] a perfectly-periodic schedule can dramatically reduce the power consumptions of clients.

Obviously not every set of clients admits a perfectly-periodic schedule. Bar-Noy et al. considered the problem of scheduling clients with different periods but equal (unit) request sizes; they showed that deciding whether a given set of clients admits a perfectly-periodic schedule is NP-hard [27]. Patt-Shamir et al. presented a tree based method of constructing perfectly-periodic schedules for unit size requests [28], [29]. In order for this method to work the different periods of the clients must be represented in a tree structure. They gave necessary conditions that a set must satisfy in order to have a tree representation; however, they showed that, as the results in [27] imply, not every schedulable set of periods can be represented by a tree.

Approximation algorithms for perfectly-periodic schedules take two approaches. The first approach aims at minimizing the average waiting time of a client; it relaxes the demand for the schedule to be perfectly-periodic and allows the intervals between allocations to vary (see e.g., [26], [30]). The second approach demands that the schedule remain perfectly-periodic but allows the actual periods to vary from the requested periods. In this approach the goal is to modify the requested periods by as little as possible (see e.g., [28], [29]).

The common assumption of nearly all works on perfectly-periodic schedules is that each request is for one time slot, i.e., $R_i = 1$. This assumption does not hold in our scheduling problem where requests (grant) may have arbitrary sizes. The case of arbitrary size requests has been studied by Patt-Shamir et al. in [31]. They considered the case where the periods are a power of two (times a common constant) and presented an algorithm that produces a perfectly-periodic schedule whenever the bandwidth demand does not exceed a certain limit. In Section 7 we derive similar results but under less restrictive assumptions.

3.3 Scheduling Flows with QoS Requirements in Data Networks

Many scheduling algorithms have been designed for network elements, such as switches and routers, with the goal of providing end to end guarantees on bit rate and delay for each flow (see [32] for a survey). The general framework is that of N sessions that arrive into a switch with a single output link. Each session i has a required bit rate r_i and possibly a maximum delay bound D_i. Packets arriving from a session are stored in a queue reserved for that session and are served in a First Come First Served (FCFS) manner. The scheduler at the switch should decide which session to serve at each time. The most widely implemented scheduling algorithms in network switches are based on either Weighted Fair Queueing (WFQ) or Round Robin (RR). Weighted Fair Queueing algorithms aim to approximate the theoretical fluid model of General Processor Sharing (GPS) introduced by Parekh [33]. Parekh proposed a packet by packet version of GPS known as PGPS which is in fact a weighted version of a Fair Queueing algorithm proposed by Demers et al. [34]. Since then many other algorithms have been proposed in order to emulate the behavior of GPS in a better way (for example, algorithm WF^{2}Q [35] has improved fairness properties) or reduce the complexity of implementing the algorithm (see e.g., [36], [37]). Round Robin (RR) has been proposed by Nagel as a simple way of allocating bandwidth [38]. Variants of RR that have been proposed as scheduling algorithms in network elements include Weighted Round Robin (WRR) [39], Deficit Round Robin (DRR) [40], and Elastic Round Robin (ERR) [41]. Another class of scheduling algorithms that received much attention are algorithms based on the Earliest Deadline First (EDF) rule (see e.g., [42] [43]). For a single link, and when the traffic is leaky-bucket controlled, it has been shown that the non-preemptive EDF algorithm is optimal (among all non-preemptive policies) in a sense that it can satisfy the largest set of delay requirements [44] [45].

Although WFQ, DRR and EDF algorithms provide bounds on the maximum queueing delay a packet may experience, they have several limitations that renders them (at least in their original version) impractical for our scheduling problem. The main drawback of these algorithms is that they do not take into account the periodic nature of the traffic and therefore cannot reserve a grant for a packet that has not been created yet. In other words, the algorithms only guarantee the delay a packet may experience from the time it has been created, which in our problem translates into guarantees on the jitter only. In addition the algorithms provide no efficient way of setting the time reference of each flow, which is a key issue in our problem. Even if we decide to use one of the algorithms (by translating delay bounds to jitter bounds), there are still several issues that should be addressed. The bounds on the delay that WFQ and RR algorithms guarantee can be improved only by increasing the bandwidth reserved for the flow. This property implies that supporting low bandwidth flows with tight delay (or jitter in our case) requirements can be problematic. This last problem has been addressed recently in several studies [46] [47].

Finally we mention work that has been done on scheduling flows with QoS requirements in DOCSIS compliant cable networks. Bandwidth allocation for non real-time applications has been considered in [48, 49, 50]. Simulations of simple scheduling algorithms for CBR flows have been presented in [51, 52]. A two phase algorithm for allocating mini-slots for real-time flows in a DOCSIS network has been presented in [53]. This algorithm, however, does not guarantee the QoS requirements of all the flows it schedules; rather, it assumes that some packets may be lost due to QoS violation. Several works suggest to solve the problem of scheduling CBR flows simply by giving these flows higher priority (e.g., [50, 54]); our results indicate that this alone cannot ensure the QoS requirements of the flows. Scheduling of Variable Bit Rate MPEG video in DOCSIS networks has been considered in [55, 56]. The problem of assigning upstream channels to cable telephony and moving calls from one channel to another, under the assumption that all calls (CBR flows) have the same parameters, is analyzed in [57]. Previous papers on QoS scheduling for DOCSIS networks provided simulation results only. Our work, to the best of our knowledge, is the first to present an analytic analysis of the problem of scheduling CBR flows in data over cable networks.

3.4 Our Results

In this subsection we summarize the subjects we covered in this study and present some of the results we obtained. The general scheduling problem is complex. We therefore focus on several interesting special cases. We start with the simplest case where all the flows have the same grant interval and progress to more complex settings.

3.4.1 Single Grant interval

In Section 4 we assume that all the flows have the same grant interval but each flow may have a different grant size. In this case the Feasible-Set and Legal-Schedule problems are trivial. The Optimal-Subset problem is solved by a polynomial algorithm when the goal is to maximize the number of scheduled flows but is NP-hard when the goal is to maximize channel utilization.

3.4.2 Two Related Grant Intervals

In Sections 5 and 6 we consider the case of two related grant intervals, i.e, intervals which are an integer multiple of each other. We show that in this case the Feasible-Set problem is already NP-complete.

Theorem 1: The Feasible-Set problem with two different grant intervals is strongly NP-complete.

We observe that in the case of related grant intervals the scheduling problem can be modelled as a variant of bin packing where bin sizes can be modified in a constrained manner. Based on this observation we develop several scheduling algorithms and investigate their performance. For

two of the algorithms, $LS - LB$ and NFJ, we derive sufficient and necessary conditions under which the algorithms produce a legal schedule.
Theorem 4: Let F be a set of n flows with two grant intervals $I_2 = m\, I_1$. Denote by S_2 the maximal grant size of flows in F_2 and by J_1 the minimal grant jitter of flows in F_1. Suppose F satisfies the following two properties:

1. $\eta = \sum_{f_i \in F} \frac{S(i)}{I(i)} \leq 1$
2. $S_2 \leq J_1 + 1$

then F is a Feasible-Set. Furthermore, both algorithms $LS - LB$ and NFJ always find a legal schedule for all flows in F.

3.4.3 Several Related Grant Intervals

In Section 7 we extend the analysis to the case where there are $K > 2$ related grant intervals. We introduce a new scheduling algorithm called $FFJ - K$ and a variant of this algorithm which we call $PP - FF$. The $PP - FF$ algorithm is used for constructing perfectly-periodic schedules, i.e., with zero jitter. We prove the following property of the algorithm.
Theorem 5: Let F be a set of flows with K related grant intervals such that $I_j = m_j\, I_{j-1}$ where m_j is a positive integer for all $2 \leq j \leq K$. Denote by S_{max} the maximal grant size of flows in F, and by $W_F \equiv \sum_{f_i \in F} S(i)/I(i)$ the total bandwidth requirements of flows in F. Algorithm $PP - FF$ always produces a perfectly-periodic schedule for a subset of flows $F' \subseteq F$ such that the channel utilization of the schedule satisfies

$$\eta = \sum_{f_i \in F'} \frac{S(i)}{I(i)} \geq \min\left\{W_F,\, 1 - \frac{S_{max} - 1}{I_1}\right\}.$$

Furthermore, no other algorithm can guarantee to produce a perfectly-periodic schedule with a higher channel utilization, i.e., with $\eta > 1 - \frac{S_{max}-1}{I_1}$.

For the case of non zero jitter we derive necessary and sufficient conditions for the $FFJ - K$ algorithm to produce a legal schedule.
Theorem 6: Let F be a set of flows with K related grant intervals such that $I_j = m_j\, I_{j-1}$ where m_j is a positive integer for all $2 \leq j \leq K$. Denote by S_{max} the maximal grant size of flows in $F \setminus F_1$ and by J_{min} the minimal grant jitter of flows in $F \setminus F_K$. Algorithm $FFJ - K$ produces a legal schedule for all flows in F whenever F satisfies the following two conditions:

1. $W_F \equiv \sum_{f_i \in F} \frac{S(i)}{I(i)} \leq 1$
2. $J \geq (K - 1)(S_{max} - 1)$

Furthermore, no other algorithm can guarantee to produce a legal schedule of all flows in F, unless F satisfies both conditions 1 and 2.

3.4.4 Unrelated Grant Intervals

In Section 8 we consider the general case where grant intervals are not necessarily related. To exploit the advantages of related grant intervals we propose to round the original set of intervals into a set of related intervals. We consider the consequences of rounding for different types of application and analyze the effects of rounding on the performance of scheduling algorithms designed for related grant intervals.

3.4.5 Online Scheduling

In Section 9 we investigate the online version of the scheduling problem where flows are established (and possibly end) over time. We demonstrate the importance of the tolerated jitter in online scheduling and present three algorithms designed for three different categories of tolerated jitter. In the first category the tolerated jitter should satisfy $J_j \geq I_j$; in this case an offline algorithm can be easily modified to serve as an online algorithm. The second category is when $J_j \geq I_1$; for this category we develop an algorithm called OLL which has the following property.

Theorem 8: Let F be a set of flows with K related grant intervals which are powers of 2 times I_1. Denote by S_{max} the maximum grant size in F and require that grant jitters in F satisfy $J_1 \geq 0$, $J_j \geq \min\{I_1,\, (K-1)S_{max},\, (2^{K-j}-1)S_{max}\}$, $\forall j \geq 2$. When algorithm Online Least Loaded (OLL) is used to online schedule the set F the channel utilization achieved satisfies

$$\eta \geq \min\left\{W_F,\, 1 - \frac{KS_{max}-1}{I_1} + \frac{K(K-1)S_{max}}{2I_K}\right\}$$

The third category we consider is that of zero jitter. We present a modification of OLL that produces schedules with zero jitter.

We present simulation results of the different algorithms and compare their worst case performance bounds to their average behavior.

3.4.6 Multi Channel Scheduling

In Section 10 we extend the analysis to consider multi channels. Since scheduling flows with two grant intervals is NP-hard even when a single channel is used, we restrict our attention to the case of uniform calls, i.e., when all the calls have the same values of I, S and J. Scheduling uniform flows on a single channel is trivial, we show that with multiple channels the problem becomes NP-hard.

Theorem 9: Deciding if a given set of uniform flows, can be scheduled on multiple channels is strongly NP-complete.

We concentrate on the online version of the problem and consider the possibility to switch flows from one channel to another. We define several scheduling algorithms and investigate their performance under several settings.

4 Scheduling Flows with a Single Grant Interval

We assume that all the flows have the same grant interval but may have different grant sizes. We denote by I_1 the grant interval and by F the set of all flows. In this case we can easily answer most scheduling problems.

1. **Feasible-Set** - A legal schedule exists if and only if the sum of grant sizes of all the flows is no more than the grant interval, i.e., if $\sum_{f_i \in F} S(i) \leq I_1$. Practically all we have to check is that the bandwidth requirement of all the flows does not exceed 1.

2. **Legal-Schedule** - A legal schedule for a feasible set is obtained by ordering the flows one after the other in a single interval and repeating this sequence in all the following intervals. The initial order of the flows is not important.

3. **Optimal-Subset** - The Optimal-Subset problem has a simple solution when the goal is to maximize the number of flows in the feasible subset. We sort the flows in increasing order of their grant sizes and keep adding flows to the subset as long as the sum of grant sizes in the subset does not exceed I_1.

   In the case where the goal is to maximize the channel utilization, Optimal-Subset is identical to the subset-sum problem. The subset-sum problem is NP-hard but can be solved in pseudo polynomial time [58]. There are also many approximation algorithms for subset-sum (see e.g., [59, 60, 61]).

Online scheduling in the case of a single grant interval is fairly simple. In the case of permanent flows the online algorithm is identical to the offline algorithm. Implementing admission control is simple, a new flow with grant size $S(i)$ can be admitted if and only if the channel utilization when the flow arrives does not exceeds $1 - S(i)/I_1$. The case of temporary flows is a little more complicated since, due to the departure of flows, the currently scheduled flow may not occupy consecutive slots. In order to admit a new flow the algorithm may have to change the locations of admitted flows (using the tolerated jitter) within the basic interval. More details about the online problem are given in Section 9.

5 Scheduling Flows with Two Related Grant Intervals

In this section we consider the offline version of the scheduling problem with two related grant intervals satisfying $I_2 = m_2 \cdot I_1$, where m_2 is an integer. We denote by $\Sigma_1 = \sum_{f_i \in F_1} S(i)$ the total size of all grants in F_1 and similarly by Σ_2 the size of all grants in F_2. We denote by n_1 and n_2 the number of flows in F_1 and F_2, respectively.

In order to produce a legal schedule it is sufficient to find a legal allocation of the flows over an interval of length I_2. Once we find such an allocation, we can repeat it over and over to get a legal schedule (see Figure 2). We are therefore concerned with the allocations over a single interval of length I_2 which we call the *basic interval.* In the basic interval each flow in F_1 must be scheduled (in a legal manner) m_2 times, while each flow in F_2 must be scheduled once. To find a legal schedule we first group all flow in F_1 and allocate them at the beginning of m_2 periodic intervals of length I_1. Once the flows in F_1 have been scheduled the free slots in the basic interval consists of m_2 gaps which are left between successive allocations of flows in F_1; the size of each gap is $I_1 - \Sigma_1$. Our task is to allocate the flows in F_2 in those gaps to form a legal schedule. An example of an allocation of flows in F_1 and F_2 over two basic intervals is shown in Figure 2.

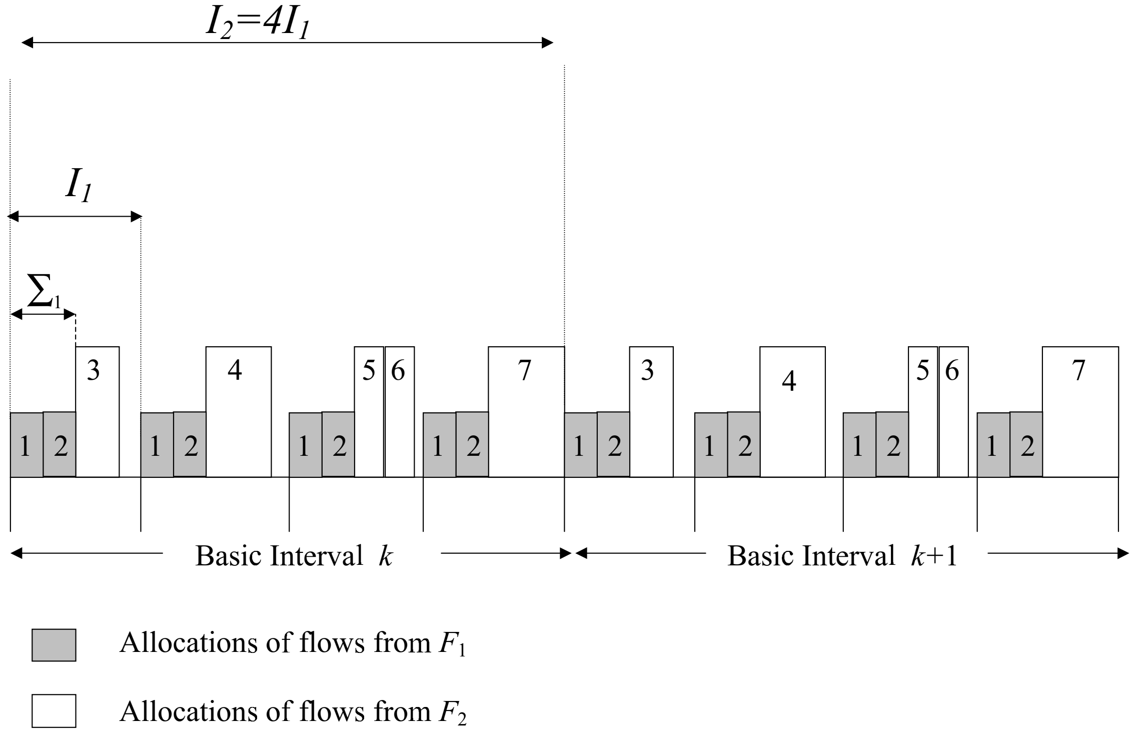

Figure 2: Example of a legal schedule of flows in F_1 and F_2 over two basic intervals.

The problem of scheduling flows in F_2 brings to mind packing problems such as bin packing or multiple knapsack. Both problems have been widely researched and investigated (see [62] and [63] for comprehensive surveys). We concentrate on the bin packing problem. In the classical one-dimensional bin packing problem, we are given a list of items $L = (a_1, a_2, ..., a_n)$, each with a size $s(a_i) \in (0, 1]$ and are asked to pack them into a minimum number of unit capacity bins. In our current scheduling problem there are m_2 bins of size $B = I_1 - \Sigma_1$, and n_2 items. Each item corresponds to a flow in F_2 and its size is the flow's grant size. The goal is to pack the items into the bins in a way that results in a legal schedule. The variant of bin packing we face in the scheduling problem has several important characteristics which make it different from classical bin packing

- The sizes of the bins may be modified in a constrained manner. This is due to the grant jitter of flows in F_1 that permits us to move them forward in time by as many slots as their grant jitter allows.
- Some items may be bigger than the bin size. Such items may still be packed by increasing the sizes of some of the bins.
- There is a fixed number of bins.

5.1 Changing Bin Sizes

The main difference between the scheduling problem and classical bin packing is the ability to change bin sizes. Initially all m_2 bins have the same size; we call this size the *nominal bin size* and denote it by B. If the jitter of all flows in F_1 is J it means that bins of sizes in the range $B - J$ to $B + J$ can be created. Clearly the total free space does not change so the total sizes of all bins must remain a constant equal to $m_2 B$. Figure 3 presents an example of how we can get bins of sizes 2 to 6 from bins of nominal size $B = 4$, when the jitter is $J = 2$. Suppose we want to change the size of bin number k. If we want to increase the bin's size, we move the group of flows in F_1 in the next bin (number $k + 1$) forward in time. If we want to decrease the bin's size, we move the flows in F_1 in bin number k forward in time. Note that it is not allowed to move the flows in F_1 to a time which is earlier than the nominal allocation time. The displacement of flows in F_1 is by at most J which means the maximum bin size we can create is $B + J$ and the minimum bin size is $B - J$. It is important to note that the sizes of the bins are strongly dependent since increasing the size of one bin results in decreasing the size of the next bin. Because of this dependency there are some sequences of bin sizes which are not feasible, i.e., there is no way to produce them. Take for example the five bins shown in Figure 3 and assume $J = 3$. It is easy to verify that the sequence $\{B_1 = 7, B_2 = 5, B_3 = 5, B_4 = 2, B_5 = 1\}$ is not legal since it violates the jitter constraint. Note that it is possible to obtain a legal sequence from the same set of bin sizes by ordering the bins; for example the sequence $\{B_1 = 7, B_2 = 1, B_3 = 5, B_4 = 5, B_5 = 2\}$ is legal. There are, however, sets of bin sizes that are not feasible under any order; in our example it is easy to see that the set $\{6, 6, 6, 1, 1\}$ is not feasible. We conclude that while it is possible to create any bin size in the range $B - J$ to $B + J$, it may be impossible to create certain sequences of bin sizes even if the bins are all in that range.

We use two basic approaches for changing bin sizes. In the first approach, which we call *state dependent*, the algorithm assumes that the bins are ordered in a certain way and this order does not change. The fixed bin order enables the algorithm to calculate the possible sizes each bin may have and thus enables it to decide whether to pack a given item or discard it. As a result the algorithm can make sure that during the packing process it is always left with a legal schedule. The second approach is called *state independent* because the algorithm does not maintain the state of the bins that determines the possible bin sizes. Instead, scheduling is divided into two stages. In the first stage the algorithm packs the items while assuming the size of each bin can change by a given constant. In the second stage the algorithm tries to order the bins that were created in the first stage so as to produce a legal schedule. In the next subsections we discuss some of the properties of each approach.

5.2 State Independent Algorithms

State independent algorithms attempt to solve the scheduling problem in two stages. In the first stage the algorithm packs the flows in F_2 into m_2 bins. The algorithm is allowed to change bin

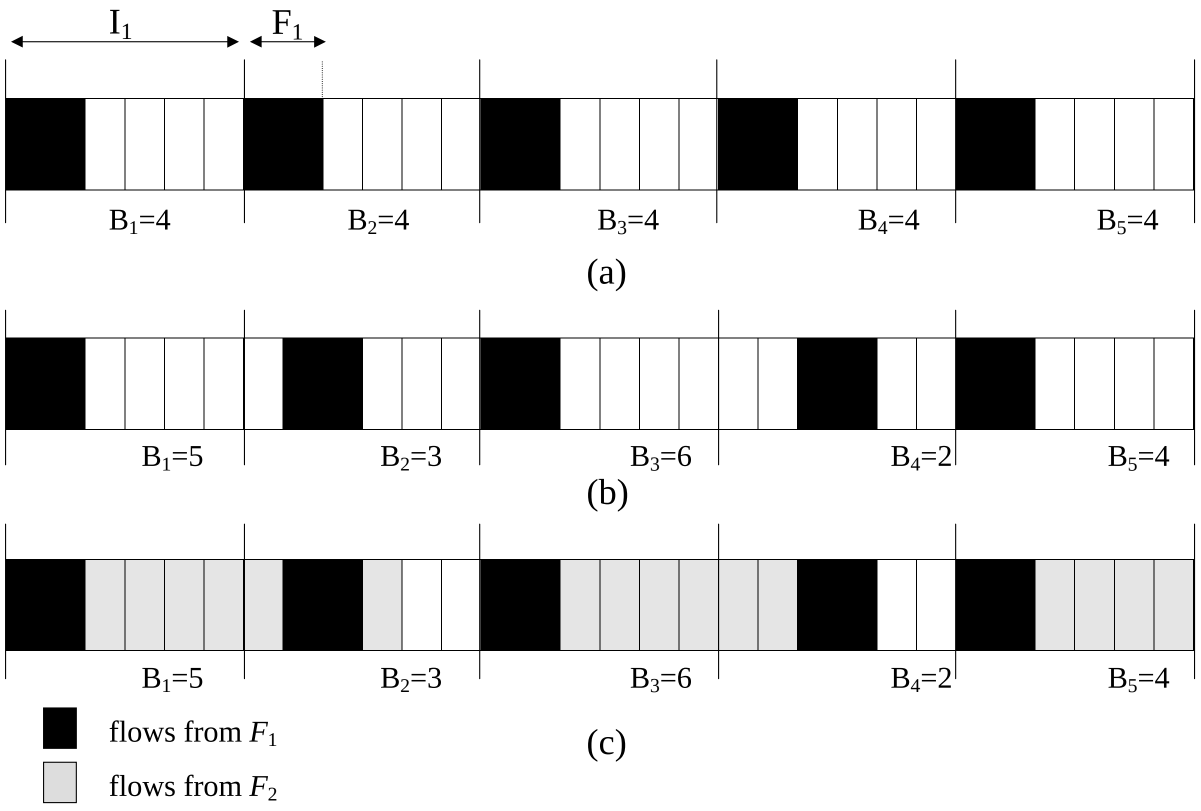

Figure 3: Example of changing bin sizes. Initially all bins are of size 4 (a). We move the flows in F_1 within the jitter limits to create bins of sizes 2-6 (b). We can now schedule flows in F_2 in the new bins (c)

sizes and when doing so the algorithm is constrained only by the tolerated jitter of flows in F_1; the question of schedule feasibility is ignored. We call this stage *Bin Packing with Jitter.* In the second stage the algorithm attempts to produce a legal schedule by ordering the bins that were created in the first stage. We call the second stage *Bin Ordering with Jitter.* We now investigate each of the stages separately.

5.2.1 State Independent: Bin Packing with Jitter

The problem we consider is a variant of bin packing where the sizes of the bins may be modified in a constrained manner. Let us first present the decision version of the problem.

Bin Packing with Jitter (BPJ):
INSTANCE: A set of m bins $B_1, \ldots, B_m$, a nominal bin size B, a list $L = \{a_1, ..., a_n\}$ of n items, each with a size $s(a_k) \in Z^+$, $1 \leq k \leq n$, and a tolerated jitter $J \in Z^+$.
QUESTION: Is there a legal packing for the list of items, i.e., such that the size of each bin B_i satisfies $B - J \leq s(B_i) \leq B + J$, the sum of sizes of all m bins is $\sum_{i=1}^{m} s(B_i) = m\,B$, and the total size of the items packed in each bin does not exceed its size $\sum_{a_k \in B_i} s(a_k) \leq s(B_i)$, $1 \leq i \leq m$.

Our first step is to determine the complexity of the problem.

Claim 5.1 *Bin Packing with Jitter is NP-complete in the strong sense.*

Proof: We show a transformation form 3-PARTITION (defined below) to a restricted version of Bin Packing with Jitter. 3-PARTITION is known to be NP-complete in the strong sense [58].

3-PARTITION:
INSTANCE: A finite set A of $n = 3k$ integers $a_1, a_2, ..., a_n$ and a bound $C \in Z^+$ such that $C/4 < a_j < C/2$ for $j = 1, ..., n$ and $\sum_{j=1}^{n} a_j = k\,C$.
QUESTION: Can A be partitioned into k disjoint subsets $S_1, ..., S_k$ such that $\sum_{j \in S_i} a_j = C$ for $i = 1, ..., k$?

Consider an arbitrary given instance of 3-PARTITION with a bound C and a set of $n = 3k$ elements $A = \{a_1, a_2, ..., a_n\}$. For a given jitter J we construct the following instance of BPJ: let the nominal bin size be $B = C + J$ and let $m = 2k$. The list of items contains $n + k$ items; the first n items are the items in A and we add k more items of size $B + J$.

In order to pack all items of size $B + J$ there must be k bins of size $B + J$. As a result the sizes of the remaining k bins must be $B - J$. After placing the items of size $B + J$ in bins of size $B + J$ we face the problem of packing the items in A into k bins of size $B - J = C$. This is exactly the 3-PARTITION problem which means that the answer to BPJ is "yes" if and only if the answer to 3-PARTITION is "yes". ■

An immediate corollary of the intractability of BPJ is the following.

Theorem 1 *The Feasible-Set problem with two different grant intervals is strongly NP-complete.*

Proof: Feasible-Set is at least as hard as bin packing with jitter since any legal solution of Feasible-Set is also legal for BPJ, but a legal solution of BPJ is not necessarily a legal solution for Feasible-Set. ■

5.2.2 State Independent: Bin Ordering with Jitter

Bin ordering is performed after a bin packing algorithm produced a set of variable size bins. The task of a bin ordering algorithm is to order the bins in a way that produces a legal schedule. To define a legal schedule, recall that the actual jitter of a grant is the difference between the actual and nominal times of the grant. In a legal schedule the actual jitter of flows in F_1, after ordering any number of bins, must not exceed the tolerated jitter; when scheduling the last bin the actual jitter must be zero. We assume there are m bins, the nominal bin size is B and the tolerated jitter of flows in F_1 is J. Let us define the problem in a formal way.

Bin Ordering with Jitter (BOJ):
INSTANCE: A nominal bin size B, a set Q of m bins, each with a size $s(B_i) \in Z^+$, $1 \le i \le m$ and a tolerated jitter $J \in Z^+$.
QUESTION: Is there a legal schedule for the set of bins, i.e., such that after ordering any number $k < m - 1$ of bins the actual jitter is at most J and after ordering all m bins the actual jitter is zero? Note that the actual jitter cannot be less than zero. Hence when ordering the k^{th} bin, its size may be increased to make the sum of sizes of the first k bins equal to kB.

Example: Suppose the nominal bin size is $B = 10$ and the tolerated jitter is $J = 5$. We are given a set of $m = 5$ bins with the following sizes: $Q = \{15, 13, 8, 8, 6\}$. Note that here, as well as in the sequel, we identify a bin by its size and not its sequence number; this is done to simplify the notation and should cause no ambiguity. We can produce a legal schedule by scheduling the bins as follows: $\{15, 8, 8, 13, 6\}$. The actual jitter during the schedule is $\{5, 3, 1, 4, 0\}$. The only other legal schedule in this example is $\{15, 6, 13, 8, 8\}$.

Our first goal is to determine the complexity of Bin Ordering with Jitter.

Claim 5.2 *Bin Ordering with Jitter is NP-complete in the strong sense.*

Proof: We show a transformation form 3-PARTITION (defined in subsection 5.2.1) to a restricted version of Bin Ordering with Jitter. 3-PARTITION is known to be NP-complete in the strong sense [58].

Consider an arbitrary given instance of 3-PARTITION with a bound C and a set of $n = 3k$ elements $A = \{a_1, a_2, ..., a_n\}$. We construct the following instance of Bin Ordering with Jitter: let the nominal bin size be $B = C + 1$ and let the tolerated jitter be $J = C$. We define $m = n + k$ bins. The sizes of the first n bins are constructed from the elements in A such that $s(B_i) = B + a_i$, $1 \leq i \leq n$. We add k more bins of unit size, $s(B_{n+1}) = ... = s(B_{n+k}) = 1$.

We now show that the two problems are equivalent. By definition the set A can be partitioned into k subsets $S_1, \ldots, S_k$ of size C if and only if the answer to 3-PARTITION is "yes". For any "yes" instance of 3-PARTITION we construct a legal schedule for Bin Ordering with Jitter by performing the following k iterations. In the first iteration we schedule three bins that were constructed from the elements in S_1; the actual jitter at this point of the schedule is $j = C = B-1$. We can therefore schedule B_{n+1}, which is of unit size, to make the actual jitter zero. We proceed in a similar way; in iteration i, $1 \leq i \leq k$ we schedule three bins that were constructed from the elements in S_i and then schedule B_{n+i}, which is of unit size, to make the actual jitter zero. At the end of iteration k we have a legal schedule since all the bins have been scheduled and the actual jitter is zero. Now consider the case where the answer to 3-PARTITION is "no". Since the sum of sizes of all the bins is exactly mB, a legal schedule can have no unused slots. Bins of unit size can, therefore, only be scheduled when the actual jitter is $B - 1$ which means that during a legal schedule there must be k points where the actual jitter is $B - 1$. Since the answer to 3-PARTITION is "no" there could be no such k subset and the answer to Bin Ordering with Jitter is therefore also "no".

We showed that for any instance of 3-PARTITION we can construct an instance of Bin Ordering with Jitter for which the "yes" and "no" instances are identical. We conclude that Bin Ordering with Jitter is at least as hard as 3-PARTITION. ■

In our proof the tolerated jitter is $J = B - 1$. We can construct a similar proof using other values of tolerated jitter. If $J = B - x$ we simply set $B = C + x$ and choose n bins of size $s(B_i) = B + a_i$ plus k bins of size x. Note however that in our example $J = C$ so when J is very small the 3-PARTITION problem can be solved easily, simply because the elements in A have a limited range of values.

5.2.3 Relation between Bin Packing and Legal Schedules

When we presented BPJ we allowed the bins to take any size in the range $B - J$ to $B + J$. It is important to note that finding a legal solution to BPJ does not mean a legal schedule exists. As an example for a legal packing that does not have a legal schedule take $B = 10$, $J = 3$ and $m = 5$ bins with the following sizes: $\{12, 12, 12, 7, 7\}$. The packing is legal since the sizes of the bins are in the range $B - J = 7$ to $B + J = 13$. However, it is easy to verify that there is no way we can order the bins to obtain a legal schedule.

In many cases we expect more than one possible solution to BPJ. It is therefore interesting to check whether there is a reason to prefer one solution over the others. Obviously if a the solution to BPJ consists of bins of size B only, it would be our preferred solution since in this case the Bin Ordering stage is not needed. What about other solutions? Is it better, for example, to select a solution where the maximum bin violation is minimal, or a solution where the number of violations is minimal, or maybe a solution with the most balanced number of bins with overflow compared to bins with underflow. While it is hard to give a clear answer to this question, the following theorem identifies a set of solutions for which we know a legal schedule exists. The theorem indicates that a good optimization function for BPJ is to try to minimize the difference between the largest bin and the smallest bin.

Theorem 2 *Given a nominal bin size B and a tolerated jitter J, let the set of m bins $Q = \{B_1, ..., B_m\}$ be some legal solution of BPJ in which $\sum_{i=1}^{m} s(B_i) = mB$. Denote the difference between the largest bin and the smallest bin by $\Delta \equiv \max_{B_i \in Q}\{s(B_i)\} - \min_{B_i \in Q}\{s(B_i)\}$. The following properties hold:*

1. *If $\Delta \leq J + 1$ a legal schedule exists.*
2. *If $\Delta > J + 1$ a legal schedule may not exist.*

Proof: We begin by proving the first property.

Case 1: $\Delta \leq J + 1$

Assume by contradiction that an optimal bin ordering algorithm cannot find a legal schedule for a certain set of bins with $\Delta \leq J + 1$. Consider the operation of the optimal algorithm as a sequential process of ordering bins. In order for the optimal algorithm to fail there must be a point where one or more slots are left unused. Let us assume that the first time this situation happens is when the k^{th} bin is ordered, that is $\sum_{i=1}^{k} s(B_i) < kB$. Assume that after ordering $k - 1$ bins the actual jitter is j, $0 \leq j \leq J$, the size of the smallest bin which is larger than B is $B + x$, $1 \leq x \leq J$ and the size of the smallest bin is $B - y$, $1 \leq y \leq J$. Ordering the k^{th} bin results in an unused slot only if the following two conditions are met:

1. $j + B + x \geq B + J + 1$ - otherwise a bin of size $B + x$ can be ordered without wasting any slots.
2. $j + B - y \leq B - 1$ - otherwise a bin of size $B - y$ can be ordered without wasting any slots.

It follows from the above conditions that $j + x \geq J + 1$ and $j \leq y - 1$, which means that $x + y \geq J + 2$. However, this is clearly a contradiction since we assume $\Delta \leq J + 1$ but by definition $\Delta \geq (B + x) - (B - y) = x + y$.

Case 2: $\Delta > J + 1$

To show that if $\Delta > J + 1$ we cannot guarantee that a legal schedule exists, consider the following example. The nominal bin size is $B = 10$, there are $m = 5$ bins with sizes $\{12, 12, 12, 7, 7\}$ and the jitter is $J = 3$. In this example $\Delta = J + 2 = 5$ and there is no legal schedule for the set of bins. ■

From Theorem 2 it is clear that the bin packing stage should try to minimize Δ. If we can find a packing with $\Delta \leq J+1$ we know a legal schedule exists. Furthermore, we show in the next section that in this case the Bin Ordering stage can be solved in polynomial time.

5.3 State Dependent Algorithms

Assume there are m bins. A state dependent algorithm maintains a state for each of them. A bin's state is made of two components: the free space in the bin, and the actual jitter of flows in F_1 (the displacement from the nominal time) in the bin. The algorithm maintains a legal schedule at all times. It uses the states of the bins to check if a given flow can be inserted into a certain bin without violating the scheduling rules. The states are updated after each flow is inserted.

We define bin B_i as starting from nominal time t_{i-1} and ending at nominal time t_i. We denote by $C(i)$ the free space in bin B_i and by $d(i)$ the displacement from t_i (the actual jitter) of flows in F_1 in B_i. The pair $(C(i),\ d(i))$ describes the state of bin B_i. In the initial state $C(i) = B$ and $d(i) = 0$ for all $1 \leq i \leq m$. When the algorithm packs an items into a bin the free space in that bin decreases. At some stage the algorithm may decide to pack an item in a bin that does not have enough free space to contain the item. In this case the algorithm tries to increase the size of that bin. We make a reasonable assumption that the algorithm does not increase the size of a bin unless it must do so in order to pack an item. Increasing the size of bin B_i by s results in increasing the displacement in the next bin $d(i+1)$ by s. Note that as a result the state of bins B_{i+1} to B_m may also change. When the algorithm decides to insert an item of size s into bin B_i it runs the INSERT procedure which is described below. If the item can be inserted, the procedure updates the state of all bins and returns SUCCESS. Otherwise the procedure returns FAIL and does not change the state of the bins.

```
Procedure INSERT($s, i, m$)
/* Insert an item of size $s$ into bin $B_i$ */
/* $m$ is the number of bins to consider */
IF ($s \leq C(i)$) {        /* item fits in $B_i$ */
   $C(i) = C(i) - s$;
   RETURN(SUCCESS);
}
ELSE {
   IF i==m RETURN(FAIL); /* the size of $B_m$ cannot be increased */
   $displace = s - C(i)$;        /* necessary displacement of flows in the next bin */
   $temp\_C(i) = 0$;
   FOR ($k = i + 1$ to $m$) {
         $temp\_d(k) = d(k) + displace$;        /* use temporary states /*
         IF ( $temp\_d(k) > J$ ) RETURN(FAIL);        /* jitter violation /*
         IF ( $displace \leq C(k)$ ) {
                  /* bin $B_k$ has enough free space for the required displacement */
              $temp\_C(k) = C(k) - displace$;
              FOR ($n = i + 1$ to $k$) {        /* update state /*
                   $d(n) = temp\_d(n)$;
                   $C(n) = temp\_C(n)$;
              }
              RETURN(SUCCESS);
         }
         ELSEIF ( $k == m$ ) RETURN(FAIL);        /* last bin /*
         ELSE {
              $displace = displace - C(k)$;        /* displacement for next bin /*
              $temp\_C(k) = 0$;
         }
   }
}
```

In some cases an algorithm may decide to remove an item from a certain bin. In this case the algorithm uses the REMOVE procedure that updates the states of the bins. We assume the call to REMOVE is legal, that is, an item of size s exists in bin B_i. If B_i has some free space (which means that the size of B_i was not increased) then removing an item simply results in increasing the free space in B_i. However, in some case removing an item from bin B_i may change the states of all bins B_i to B_m.

```
Procedure REMOVE($s, i, m$)
/* Removes an item of size $s$ from bin $B_i$ */
/* $m$ is the number of bins */
IF ($i == m$ or $C(i) > 0$ or $s == 0$) {        /* last update */
  $C(i) = C(i) + s$;
  RETURN;
}
ELSE {
      $C(i) = \max\{s - d(i+1), 0\}$        /* update free space in $B_i$ */
      $displace = \min\{s, d(i+1)\}$
      $d(i+1) = d(i+1) - displace$
      /* update displacement in $B_{i+1}$ */
      REMOVE($displace$, $i+1$)
}
```

The main advantage of using a state dependent algorithm is that it always produces a legal schedule. Maintaining the state is simple and requires at most $O(m)$ operations per item. It is easy to create a state dependent version of most standard bin packing algorithms. For example, we can use the First Fit algorithm [64] which goes over the bins starting from B_1 and packs an item in the first bin in which it fits. If the algorithm cannot pack the item in any of the bins the item is discarded. The main disadvantage of state dependent algorithms is that they make no effort to order the bins and can therefore perform poorly in some cases. Consider for example the following list of item sizes $L = \{8, 8, 6, 6, 6, 6\}$ with $B = 10$, $J = 2$, and $m = 4$. Any algorithm that packs the items of size 8 in bins other than B_2 and B_4 will fail to pack the whole list. This however is the case for most standard bin packing algorithms such as Next-Fit, First-Fit or Best-Fit.

Finally we note that, as we shall see in the sequel, state dependent algorithms are the preferred choice when there are more than two grant intervals. State independent algorithms are problematic when there are more than two grant intervals since they assume that the only constraint on the order of the bins is the tolerated jitter; however, when there are more than two grant intervals there are additional constraints on the order of the bins.

6 Approximation Algorithms for Two Related Grant Intervals

In this section we describe several approximation algorithms for scheduling flows with two related grant intervals. We evaluate the performance of an algorithm as the ratio between the total size of items the algorithm packs and the total size of items an optimal algorithm can pack. For a given list L of n items and algorithm A, let $A(L)$ be the total size of items from L that algorithm A packs in m bins, let $OPT(L)$ denote the total size of items from L an optimal algorithm can pack. We now define the performance ratio of A using equations (1)-(3). If the goal is to pack the maximum number of items the definition of the performance ratio remains the same; we simply let $A(L)$ and $OPT(L)$ represent the number of packed items.

Bin packing problems (in contrast to knapsack problems) assume that all items must be packed and there is no effort to optimize the selection of a feasible subset. As a result the performance of any bin packing algorithm is going to be primarily effected by the subset selection. We make the following assumption to avoid the effect of item selection:
Compact List Assumption: For a scheduling problem with m bins and nominal bin size B, the sum of sizes of all items is no more than the sum of sizes of all bins, that is $\sum_{i=1}^{n} s(a_i) \le m\,B$.

6.1 State Independent Scheduling Algorithms

A state independent scheduling algorithm can be constructed by combining two algorithms. In the first stage we run a bin packing with jitter approximation algorithm. In the second stage we try to produce a legal schedule by running a bin ordering with jitter approximation algorithm. If the bin ordering algorithm fails to schedule all the bins, we may also add a third stage in order to improve the schedule, i.e., to maximize the total sum of items in the schedule. In this stage we keep discarding items from the list and then attempt to schedule the new set of items. We repeat the process until the best schedule is found. Since we know that when $\Delta \le J + 1$ a perfect-schedule can always be found, one possible rule for discarding items may be to discard the smallest item that results in decreasing Δ. The basic template for such a state independent scheduling algorithm is the following:

Template for a State Independent Algorithm

1. **Bin Packing**: Use a BPJ approximation algorithm to pack the flows into bins.
2. **Bin Ordering**: Use a BOJ approximation algorithm to order the bins.
3. **Improve**: Repeat until the new order does not provide an improved schedule
   (a) Discard the smallest item that results in decreasing Δ.
   (b) Run the BOJ algorithm on the new set of bins.

In the next subsections we describe several algorithms for BPJ and BOP.

6.1.1 Approximation Algorithms for Bin Packing with Jitter

We showed that the decision version of Bin Packing with Jitter is NP-complete. We now define an optimization version of the problem. The optimization function is either to maximize the total size of the packed items or to maximize the number of packed items.

INSTANCE: A set of m bins, a nominal bin size B, a list $L = \{a_1, ..., a_n\}$ of n items, each with a size $s(a_k) \in Z^+$, $1 \le k \le n$ and a tolerated jitter $J \in Z^+$.

GOAL: Find a set $L' \subseteq L$ with the maximum total size of items $\sum_{a \in L'} s(a)$ (or maximum number of items) such that L' has a legal packing, i.e., the size of each bin B_i ($1 \leq i \leq m$) satisfies $B - J \leq s(B_i) \leq B + J$, and the total sizes of all bins satisfies $\sum_{i=1}^{m} s(B_i) \leq m\,B$.

Remark: We define the size of a bin to be the maximum between the bin's content (sum of packed items) and $B - J$.

The first algorithm we present for BPJ is a modification of the well known List Scheduling (LS) algorithm. The LS algorithm was presented by Graham in 1966 [65] as an algorithm for scheduling jobs on m identical parallel machines, with the goal of minimizing the makespan. The LS rule is to assign the next job to the least loaded machine; in the case of BPJ this rule corresponds to assigning the next item to the minimal-content bin. To apply LS to BPJ the only modification we introduce is enforcing the constraint on bin sizes. We note that LS is an online algorithm; the offline version of the algorithm is usually referred to as the Largest Processing Time (LPT) algorithm [66]. The LPT algorithm first sorts the jobs in decreasing order and then applies the LS algorithm on the sorted list.

Algorithm LS

1. Go over the items according to their order in the list.
2. Assign an item to the minimal-content bin provided that after the assignment the following two conditions are met
   - The content of the bin is no more than $B + J$.
   - The sum of all bin sizes is no more than $m\,B$.
3. Discard the item otherwise.

Note that the LS algorithm does not consider the nominal bin size in its packing decisions. The algorithm places the first m items in m different bins and may therefore perform poorly in some cases. To provide more efficient algorithms we combine LS with a standard bin packing algorithm. The algorithm starts by packing the items into bins of size B as in standard bin packing. When an item does not fit in any of the bins it is assigned according to the LS algorithm. As an example we present the combination of Best Fit Decreasing (BFD) with LS.

Algorithm BFD_{LS}

1. Order the items according to decreasing order of their sizes.
2. Try to assign each item according to the following rules:
   - If the item can be packed in one of the bins without increasing its size to more than B, assign the item according to the BF rule.
   - Otherwise, assign the item to the minimal-content bin provided that after the assignment
     - The content of the bin is no more than $B + J$.
     - The sum of all bin sizes is no more than $m\,B$.
3. If the item cannot be assigned according to the above rules, discard it.

Instead of analyzing each algorithm we presented separately, we derive a more general result that relates the tolerated jitter in the problem to the performance of an algorithm for BPJ. Denote by A_J an algorithm capable of changing a bin's size by at most $J \geq 0$ units. We are interested to find the improvement an optimal algorithm can achieve by changing bin sizes. To that end we define OPT_0 (the optimal algorithm that uses only bins of size B) to be the algorithm we analyze and OPT_J to be the optimal algorithm. We define the performance ratio $R^{\infty}_{OPT_0}$ in the conventional way using equations (1)-(3). It is easy to show that when items may be larger than B the performance ratio is not bounded. Take for example a list of items of size $B+1$, OPT_0 cannot pack any item while OPT_J can pack some of the items (up to $m-1$) by increasing the size of several bins. It is therefore more appropriate to consider the case where every item fits in a nominal bin. The following theorem relates J to the maximum improvement in the performance of an algorithm.

Theorem 3 *For the problem of BPJ with item sizes bounded by B and tolerated jitter $J < B$, the asymptotic worst case performance ratio of OPT_0 is $R^{\infty}_{OPT_0} = \frac{2J+1}{J+1}$.*

Proof: To prove the upper bound we show that the ratio between the number of bins required by OPT_0 and OPT_J to pack any list L is no more than $\frac{2J+1}{J+1}$.

Claim 6.1 *If OPT_J packs a list L in m bins, OPT_0 can pack L in $\frac{2J+1}{J+1}m$ bins.*

Proof: We start with the packing of OPT_J and create a packing that uses only bins of size B by replacing every bin which is bigger than B by two bins of size B. The content of the original bin is split between the two bins. Note that this is always possible since the size of the original bin is at most $B + J < 2B$ and it does not contain any item of size bigger than B. Since the total size of all bins must be mB, the number of large bins OPT_J can use is at most $\frac{J}{J+1}m$ (for every J bins larger than B one bin must be smaller than B). The number of bins in the packing we created is at most $m + \frac{J}{J+1}m = \frac{2J+1}{J+1}m$; the number of bins required by OPT_0 can only be smaller. ■

The upper bound of Theorem 3 follows from Claim 6.1. Out of the $\frac{2J+1}{J+1}m$ bins we pick m bins with the highest content. The total content of the bins in OPT_J is at most $\frac{2J+1}{J+1}$ times the total content of the bins in OPT_0.

We now prove the lower bound. Assume the bin size is an odd number $B = 2b-1$. We choose a list with $n = 2m$ items of size b. OPT_0 can pack only one item per bin so the total content in m bins is $OPT_0(L) = m \cdot b$. OPT_J packs the items in the following way: The first J bins are of size $B+1$ and contain two items, bin $J+1$ is of size $B-J$ and contains one item. The rest of the bins repeat this pattern. The total content of OPT_J is therefore

$$OPT_J(L) = \frac{J}{J+1}m \cdot 2b + \frac{1}{J+1}m \cdot b = \frac{2J+1}{J+1}m \cdot b.$$

■

Theorem 3 tells us that, when every item fits in a bin, the improvement we can get by using the jitter to modify bin sizes is by at most a factor of 2. This is reasonable since we know that the optimal packing is guaranteed to fill the bins to at least half of their capacity. Note that while the maximum ratio between OPT_0 and OPT_J is given in Theorem 3, it is easy to show that for any value of $J < B$, there exist a list L for which $OPT_J(L) = OPT_0(L)$; a list where all items are of size B is one trivial example.

6.1.2 Approximation Algorithms for Bin Ordering with Jitter

Since Bin Ordering with Jitter is NP-complete we are interested in approximation algorithms for the problem. Let us first define the optimization version of the problem.
INSTANCE: A nominal bin size B, a tolerated jitter $J \in Z^+$ and a set Q of m bins, each with a size $B - J \leq s(B_i) \leq B + J$, $1 \leq i \leq m$.
GOAL: Find a set $Q' \subseteq Q$ such that Q' has a legal schedule in m bins of nominal size B and Q' has the maximum sum of bin sizes.

We call a legal schedule of all m bins a *perfect-schedule.*

Our first approximation algorithm is called Largest Bin (LB). The algorithm always tries to schedule the largest possible bin without violating the jitter constrain. The second algorithm is called Minimize Actual Jitter (MAJ) and it tries to minimize the actual jitter in each step. For both algorithms Q is the list of m bins the algorithm must order. We add a bin B_{m+1} of size $s(B_{m+1}) = (-1)$ which serves as an indicator; if B_{m+1} is scheduled by the algorithm it means that the algorithm cannot find a perfect-schedule for the set Q.

Algorithm Largest Bin (LB)

```
$j_0 = 0$;   $P = \emptyset$;   $Q = \{B_1, ..., B_{m+1}\}$;   perfect=YES;
FOR $i = 1$ to $m$
    IF $i < m$  $P_i \leftarrow$ largest bin $B_k \in Q$ that satisfies $s(B_k) + j_{i-1} \leq B + J$;
    IF $i == m$ $P_m \leftarrow$ largest bin $B_k \in Q$ that satisfies $s(B_k) + j_{i-1} \leq B$;
    IF $s(P_i) == (-1)$
        perfect=NO;
        $j_i = 0$;
    ELSE
        $Q \leftarrow Q - P_i$;
        $P \leftarrow P + P_i$;
        $j_i = max\{s(P_i) + j_{i-1} - B, 0\}$;
IF perfect==YES RETURN(The bins in $P$ provide a perfect-schedule);
ELSE RETURN(No perfect-schedule was found. Bins in $P$ provide an approximation.);
```

Example: For $B = 10$ and $J = 5$ consider a set of $m = 7$ bins with the following sizes: $Q = \{15, 13, 13, 8, 8, 7, 6\}$. The LB algorithm finds the following perfect-schedule for the set: $\{15, 8, 8, 13, 7, 13, 6\}$.

Claim 6.2 *When algorithm Largest Bin terminates, the bins in P provide a legal schedule. The legal schedule in P may not be optimal.*

Proof: In each iteration i the algorithm makes sure that the actual jitter j_i of the bins in P does not exceed J. Therefore scheduling the bins according to the order in which they were entered to P, provides a legal schedule.

To show that the legal schedule in P may not be optimal we present an example that holds for any $J \geq 3$ and $B \geq J + 1$. Take $m = 6$ bins of sizes $Q = \{B + J,\ B + \lfloor \frac{J}{2} \rfloor + 1,\ B + \lceil \frac{J}{2} \rceil,\ B - 1,\ B - J,\ B - J\}$. Algorithm Largest Bin schedules the bins in P as follows: $LB = \{B + J,\ B - 1,\ B - J,\ B + \lfloor \frac{J}{2} \rfloor + 1,\ B - J\}$. The remaining bin, of size $B + \lceil \frac{J}{2} \rceil$, does not fit and the algorithm fails to produce a perfect-schedule. However, there is a perfect-schedule to the set, $OPT = \{B + J,\ B - J,\ B + \lfloor \frac{J}{2} \rfloor + 1,\ B - 1,\ B + \lceil \frac{J}{2} \rceil,\ B - J\}$. For example, for $B = 10$ and $J = 3$ we choose the set $Q = \{13, 12, 12, 9, 7, 7\}$ and for $J = 4$ the set $Q = \{14, 13, 12, 9, 6, 6\}$. ■

Claim 6.3 *If $\Delta \leq J+1$ and $\sum_{i=1}^{m} s(B_i) \leq mB$, algorithm Largest Bin produces a perfect-schedule.*

Proof: Clearly if $\sum_{i=1}^{m} s(B_i) > mB$ there can be no perfect-schedule, i.e., a legal schedule of all m bins. The proof is similar to that of Theorem 2. Let us first assume (as in Theorem 2) that $\sum_{i=1}^{m} s(B_i) = mB$. Algorithm LB tries to maximize the actual jitter in each step. Assume that after ordering k bins the actual jitter is j, the size of the smallest bin which is larger than B is $B+x$, $1 \leq x \leq J$, and the size of the smallest bin is $B-y$, $1 \leq y \leq J$. Since $x+y \leq \Delta \leq J+1$ the algorithm can always schedule a bin without leaving unused slots. As a result, for any $k \leq m$ the sum of sizes of the first k bins is at least kB. Since there are no unused slots the algorithm schedules all m bins and produces a perfect-schedule.

To show that the claim holds when $\sum_{i=1}^{m} s(B_i) < mB$, note that we can increase the sizes of the bins until the sum of sizes is mB. The way in which we increase the sizes is not important as long as we do not increase Δ (one way of doing it is to keep increasing the smallest bin by one unit). We showed that algorithm LB can find a perfect-schedule to the modified set. To obtain a perfect-schedule for the original set we use the same schedule; all we have to do is simply to decrease the bin sizes to their original size. ■

It is possible to design a similar algorithm which we call Minimize Actual Jitter. The algorithm selects the next bin such that the violation of the bin capacity is minimal. If it is not possible to fill a bin even by using the tolerated jitter the algorithm selects the largest bin. Recall that as in LB we add an indicator bin of size $s(B_{m+1}) = -1$.

Minimize Actual Jitter (MAJ)

```
$j_0 = 0$;   $P = \phi$;   $Q = \{B_1, ..., B_{m+1}\}$;   perfect=YES
FOR $i = 1$ to $m$
    $P_{over} \leftarrow$ smallest bin $B_k \in Q$ that satisfies $B < s(B_k) + j_{i-1} \leq B + J$;
    IF $P_{over} \neq \phi$  $P_i = P_{over}$;
    ELSE $P_i \leftarrow$ largest bin $B_k \in Q$ that satisfies $s(B_k) + j_{i-1} \leq B$;
    IF $s(P_i) == (-1)$
        SET perfect=NO;
        $j_i = 0$;
    ELSE
        $Q \leftarrow Q - P_i$;   $P \leftarrow P + P_i$;
        $j_i = max\{s(P_i) + j_{i-1} - B, 0\}$;
IF perfect==YES RETURN(The bins in $P$ provide a perfect-schedule);
ELSE RETURN(No perfect-schedule was found. Bins in $P$ provide an approximation.);
```

The main advantage of algorithm MAJ over LB is that it makes fewer changes to the schedule of flows in F_1. Since the two algorithms produce different schedules, we can combine them to get an improved algorithm. To do so, we simply run both algorithms and select the better solution. For example, MAJ finds a perfect-schedule for the set of bins we presented in the proof of Claim 6.2, while LB does not. In that example, if $J = 4$ and the set is $Q = \{14, 13, 12, 9, 6, 6\}$ MAJ produces the perfect-schedule $MAJ = \{12, 9, 13, 6, 14, 6\}$. The opposite is also true, that is, there are sets for which LB can find a perfect-schedule while MAJ fails. Take for example the set $Q = \{13, 11, 8, 8\}$ with $B = 10$ and $J = 3$. We can get a similar example for any $J \geq 3$ and $B \geq 4$ by choosing $Q = \{B+J,\, B+1,\, B - \lfloor \frac{J}{2} \rfloor,\, B - \lceil \frac{J}{2} \rceil\}$

As we expect, it is also possible to find a set for which a legal schedule exists but both LB and MAJ fail. Take for example the set $Q = \{18, 18, 15, 15, 9, 8, 8, 3, 3, 3\}$ with $B = 10$ and $J = 8$. LB produces the packing $LB = \{18, 9, 8, 8, 15, 3, 15, 3, 18\}$ and fails to pack a bin of size 3. MAJ

produces the packing $MAJ = \{15, 8, 8, 9, 15, 3, 18, 3, 3\}$ and fails to pack a bin of size 18. A legal schedule is given by $OPT = \{18, 3, 9, 18, 3, 15, 8, 8, 15, 3\}$

Finally, we note that other algorithms for bin ordering with jitter are possible. For example, instead of trying to minimize the actual jitter as in MAJ, an algorithm may try to minimize Δ in each step provided that no slots are wasted.

6.2 State Dependent Scheduling Algorithms

We can convert almost any standard bin packing algorithm into a state dependent scheduling algorithm. The algorithm decides in which bin the item is to be packed but the item is not packed if packing it violates the jitter constraint. If packing an item fails the algorithm may try to pack it in a different bin or discard it. As an example consider the State Dependent First Fit Decreasing (*SD-FFD*) algorithm

Algorithm *SD-FFD*

1. Order the items in decreasing order of their sizes.
2. For each item, go over the bins according to their order and pack the item into the first bin in which it fits, increasing the bin's size if necessary.
3. If the item does not fit in any of the bins discard it.

6.2.1 The Next Fit with Jitter Algorithm

We now introduce the Next Fit with Jitter (NFJ) algorithm which is a modification of the well-known Next Fit (NF) algorithm. The NF algorithm keeps only one open bin and packs items, according to their order, into the open bin. When an item does not fit in the open bin, the bin is closed, a new bin is opened and the item is packed in the new bin. Like NF, algorithm NFJ is an online, bounded space algorithm with $O(n)$ running time. Since the algorithm uses only one open bin it is very natural to design it as a state dependent algorithm. The state is maintained only for the open bin but this ensures that scheduling the bins according to the order they were closed provides a legal schedule.

In the following algorithm we denote by $c(B_i)$ and $s(B_i)$ the content and size of bin B_i, respectively. The value of j is the maximum amount by which the size of the next open bin can be increased.

Algorithm *NFJ*

```
1. Set $i = 1$.     /* $i$ holds the index of the open bin */
2. Set $j = J$.     /* $j$ holds the increase to the size of the next bin /*
3. Open bin $B_i$ and modify its size to be $s(B_i) = B + j$.
4. Pack items into $B_i$ until the next item does not fit in $B_i$.
5. Set $j = \min\{s(B_i) - c(B_i),\ J\}$.
6. Close $B_i$ and set its size to $s(B_i) = \max\{c(B_i),\ B - J\}$.
7. Set $i = i + 1$.
8. IF $i == m$ set $j = j - J$.     /* adjust $j$ for the last bin */
9. ELSEIF $i \leq m$, return to step 3.
10. ELSE STOP.
```

We begin by analyzing the worst case performance of the NFJ algorithm. We show that, similar to NF, the worst case performance ratio of NFJ is 2. Note however that this result cannot be deduced from the analysis of NF since for NFJ we allow item sizes to be larger than B.

Claim 6.4 *The asymptotic worst case performance ratio of algorithm NFJ is* $R^{\infty}_{NFJ} = 2$.

Proof: Let B_k, $1 \leq k \leq m-1$ be an arbitrary bin. When B_k is opened its size is set to be $s(B_k) = B + j \geq B$. Assume the bin is closed when an item of size x does not fit in it. If $c(B_k) \leq B$ the item is packed in B_{k+1} and we have $c(B_k) + c(B_{k+1}) \geq c(B_k) + x > s(B_k) \geq B$. If $c(B_k) + x > 2B + J$ the item is packed in B_{k+2} leaving B_{k+1} empty. However, in this case $c(B_k) > B$. It follows that the content of any two consecutive bins is at least $B+1$. We conclude that the average bin content is at least $B/2$ which gives us the upper bound on the asymptotic performance ratio.

We now show an example that proves the lower bound. In our example there are $m = k+1$ bins of size $B = 2b$, the list of items L has $2k$ items of size b, and $2k$ items of size $J+1$; the items are ordered alternately $L = \{b, J+1, b, J+1, ...b, J+1\}$. We choose $B > 2k(J+1)$ which means that all items of size $J+1$ fit in one bin, hence $OPT(L) = 2k\,(b + J + 1)$. Algorithm NFJ packs two items in each bin, one of sizes b and one of size $J+1$, hence $NFJ(L) = (k+1)(b+J+1)$. The ratio in this example is $R_{NFJ}(L) = \frac{2k}{k+1}$; the asymptotic worst case performance ratio is therefore $R^{\infty}_{NFJ} = 2$. ■

Note that in order to prove the lower bound we assumed that $B \gg J$. If this assumption does not hold the performance of NFJ improves as J increases. In particular when all items are smaller than B and $J \geq B$ the asymptotic worst case performance ratio is 1.

6.2.2 Average Case Analysis of *NFJ*

To analyze the average case performance of NFJ we use a technique which we developed for average case analysis of bounded space bin packing algorithms. The full description of this technique can be found in [67, 68] and is omitted from this paper.

We assume bins of equal size B and i.i.d. item sizes taken from some discrete probability distribution H where $h_i = Pr(s(a) = i)$, $1 \leq i \leq B$. In the analysis of the NF algorithm in [68] we used the content of the open bin as the state of the Markov chain. In the case of NFJ this information alone is insufficient because each bin's size may be modified, when it is opened or closed. We must know both the content and the size of the open bin in order to calculate the overhead when the bin is closed. The first approach is therefore to choose the pair of the content and size of the open bin as the state of the Markov chain. In this case there are $(B + J) \times (J + 1)$ states. It is possible to reduce the number of states by using the remaining space in the open bin as the state of the Markov chain. If we do so, the size of the open bin is not required. We denote by $N_t = s$ the state of the algorithm after t items were packed. The possible values of the remaining space s, are $0 \leq s \leq B - 1$ (ignoring the special case of the first bin).

We first construct the transition matrix P. Assume the algorithm is in state $N_{t-1} = s$ and the next item to be packed is of size i, $1 \leq i \leq B$. We distinguish between two cases

1. $i \leq s$: In this case the item fits in the open bin. The next state is therefore $N_t = s - i$.
2. $i > s$: In this case the item does not fit in the open bin, therefore a new bin is opened. The size of the new bin can be at most $B + J$ and its actual size depends on s. The new state is therefore
   - $N_t = B + J - i, \quad \text{if } s \geq J.$
   - $N_t = B + s - i, \quad \text{if } s < J.$

We use the above rules to construct the transition matrix P. Once we have P we can calculate the equilibrium probability vector Π, where Π_s is the stationary probability that the Markov chain is in state s. The next step is to calculate the overhead in each state, i.e., the number unused units which are left in a bin that is closed when the algorithm is in some given state. Note that overhead units are added only if the size of the next item is larger than the remaining space in the open bin and only if the remaining space is larger than J. We therefore have

$$oh_i(s) = \begin{cases} 0 & i \leq s \quad \text{or} \quad s \leq J \\ \\ s - J & J < s < i \end{cases} \tag{4}$$

For item size distribution H with an average item size $\overline{h}$, the average combined size of the items $I_{NFJ}(H)$ is calculated using the following expression:

$$I_{NFJ}(H) = \overline{h} + \sum_{s=0}^{B-1} \Pi_s \cdot \sum_{i=1}^{B} h_i \cdot oh_i(s) \tag{5}$$

The expected asymptotic performance ratio of NFJ is calculated from the ratio between $I_{NFJ}(H)$ and $I_{OPT}(H)$. Where $I_{OPT}(H)$, the average combined size of items in the optimal packing, is usually known in advance.

$$\overline{R}^{\infty}_{NFJ}(H) = \frac{I_{NFJ}(H)}{I_{OPT}(H)} \tag{6}$$

Let us consider a discrete uniform distribution, denoted by $\{B, B\}$, in which $h_i = \frac{1}{B}$, $\forall 1 \leq i \leq B$. For this distribution we can calculate the average overhead in state s

$$OH(s) = \sum_{i=1}^{B} h_i \cdot oh_i(s) = \max\left\{\frac{(B - s)(s - J)}{B}, 0\right\} \tag{7}$$

Note that since $s \leq B-1$ the total overhead is zero whenever $J \geq B-1$ and as a result the expected performance ratio is one. Figure 4 presents the asymptotic expected performance ratio of the NFJ algorithm for the $\{B, B\}$ distribution when the bin size is $B = 50$ and the jitter is in the range $0 \leq J \leq B$. As we expect the performance ratio is decreasing with J and approaches 1 as J gets closer to B. Figure 5 presents the performance ratio of NFJ for several values of jitter and values of bin size in the range $1 \leq B \leq 100$. We can see that the performance ratio is increasing with B and decreasing with J.

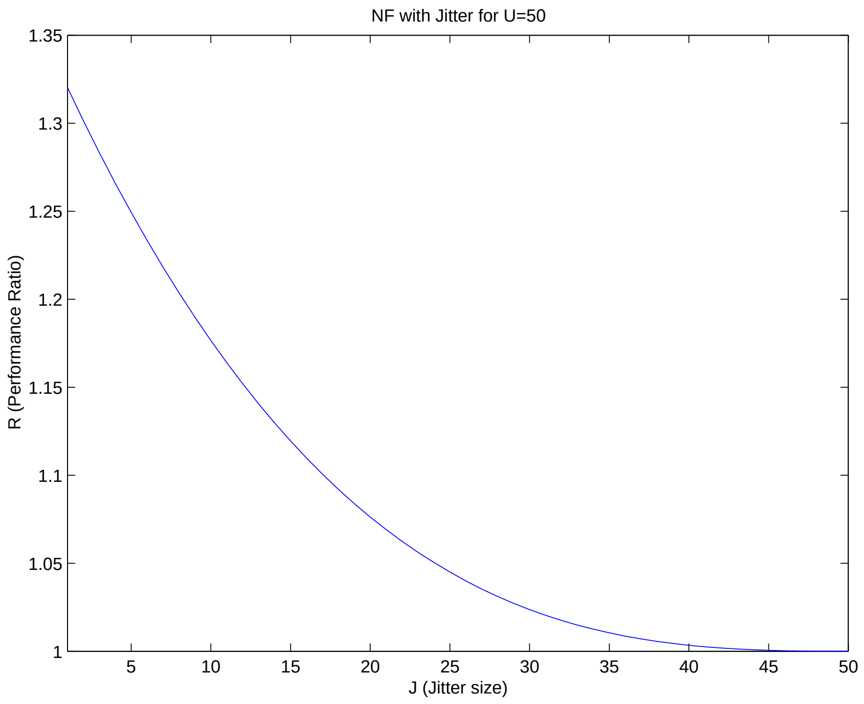

Figure 4: Asymptotic expected performance ratio of NFJ for distribution $\{B, B\}$ for $B = 50$.

6.3 Legal Schedule when Sizes are Smaller than the Jitter

In this section we show that when the maximal grant size in F_2, S_2, is no more than the minimal grant jitter in F_1, J_1, we can always find a legal schedule. The property that $S_2 \leq J_1$ is common in many real-time multimedia applications. We show that even simple algorithms are optimal under the above assumption. We choose NFJ as a state dependent algorithm. For a state independent algorithm we define an algorithm called $LS - LB$. The $LS - LB$ algorithm is a combination of the LS algorithm for BPJ and the LB algorithm for BOJ (the definitions of these algorithms appear in subsection 6.1).

Theorem 4 *Let F be a set of n flows with two grant intervals $I_2 = m_2 I_1$. Denote by S_2 the maximal grant size of flows in F_2 and by J_1 the minimal grant jitter of flows in F_1. Suppose F satisfies the following two properties:*

1. $W_F = \sum_{f_i \in F} \frac{S(i)}{I(i)} \leq 1$
2. $S_2 \leq J_1 + 1$

then F is a feasible set. Furthermore, both algorithms $LS - LB$ and NFJ always find a legal schedule for all flows in F.

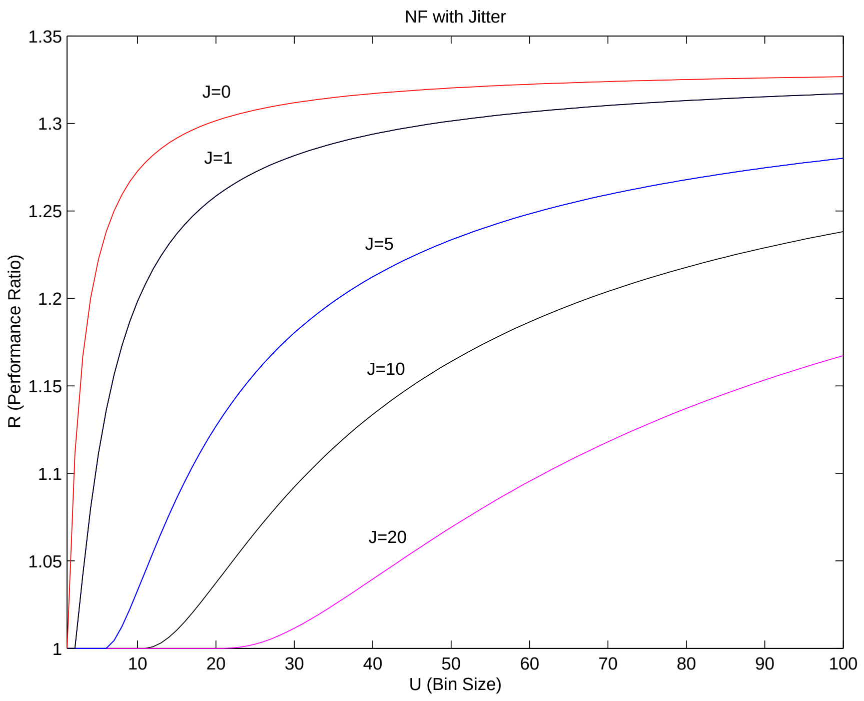

Figure 5: Asymptotic expected performance ratio of NFJ for different values of J, for $B = 100$ and distribution $\{B, B\}$.

Proof: We start by scheduling the flows in F_1 over an interval of length I_2 creating m_2 bins of size B. We then apply one of the algorithms to schedule the flows in F_2. We consider each algorithm separately.

The $LS - LB$ Algorithm

The LS algorithm assigns each item to the bin with the minimal content; therefore, at any stage of the packing the difference between the contents of any two bins is at most S_2. Since (due to condition 1) $\sum_{f_i \in F_2} S(i) \leq m_2 B$, LS does not discard any flows in F_2. It follows that when LS terminates all flows have been packed and the difference between the contents of the largest and smallest bins satisfies $\Delta \leq S_2 \leq J + 1$. From Claim 6.3 it follows that applying algorithm LB to the bins created by LS always produces a perfect-schedule.

The NFJ Algorithm

We show that whenever NFJ closes a bin, the bin does not contain unused space. Recall that a bin is defined as the gap between two consecutive blocks of flows in F_1 and that the actual jitter when a bin is opened is the difference between the nominal and actual allocation times of the block of flows in F_1 that starts the bin. When NFJ opens bin number k the actual jitter of the bin may be in the range $0 \leq j_k \leq J$. The real bin size is $s(B_k) = B - j_k$ but the algorithm modifies the size of the bin to be $s(B_k) = B + J - j_k$. Assume B_k is closed when the algorithm tries to pack an item of size y but the item does not fit since $c(B_k) + y > B + J - j_k$. It follows that the content of the bin satisfies $c(B_k) > B + J - y - j_k$. Since $y \leq J + 1$ we have $c(B_k) \geq B - j_k$ which means that the content of the bin is at least the original bin size and the bin does not contain free space.

The property of no wasted space holds for any bin which is closed by NFJ. The algorithm can therefore schedule any set of flows with bandwidth requirements not exceeding unity. Condition 1 ensures that F is such a set. ■

To see that condition 2 the Theorem is tight consider an example where $S_2 = J_1 + 2$. We choose a set F consisting of one flow in F_1 with $S(1) = 2$, $I(1) = 10$ and $J(1) = 3$, and eight flows in F_2 with $S(i) = 5$, $I(i) = 50$ and $J(i) = 3$, for $2 \leq i \leq 9$. After allocating f_1 there are five bins of size $B = 8$ to which the flows in F_2 must be assigned. It is easy to verify that only seven flows can be legally scheduled, hence F is not a feasible set.

7 Scheduling Flows with Several Related Grant Intervals

In this section we extend the analysis to the case where there are more than two related grant intervals. We are given a set F of n CBR flows; the flows are to be scheduled over a slotted TDMA channel. The set F is assumed to contain K different grant intervals which we denote by $I_1 < I_2 < \ldots < I_K$. The different grant intervals are an integer multiple of each other, that is, they satisfy $I_j = m_j \cdot I_{j-1}$, for every $2 \leq j < K$, where m_j is a positive integer. Recall that each flow f_i is characterized by its grant size $S(i)$, grant interval $I(i)$, and tolerated jitter $J(i)$. The flows are divided into K disjoint groups $F_1, \ldots, F_K$ according to their grant intervals such that $F_j = \{f_i : \ I(i) = I_j\}$.

Note that in order to produce a legal schedule it is sufficient to find a way to allocate the flows over the longest interval, I_K, which we call the *basic interval.* Once we find a legal allocation over the basic interval, we can repeat it over and over to get a legal schedule. Let us first apply the technique we developed for $K = 2$ to the case of $K = 3$. We first allocate flows in F_1 thus creating $m = m_2 \cdot m_3$ bins, each of size I_1; the free space in all the bins is $I_1 - \Sigma_1$. Flows in F_2 are then allocated into these bins. In the next step we proceed to allocate flows in F_3. The main difference is that after the flows in F_2 have been allocated, different bins may have a different amount of free space left in them. We therefore say that scheduling flows in F_3 defines a variable size bin packing problem (as opposed to uniform size bin packing in the case of $K = 2$). The problem is basically the same for $K > 3$; after allocating flows in F_j we should allocate flows in F_{j+1} into variable size bins.

Similar to the case of two intervals, bin sizes can be modified. However, when $K \geq 3$ each bin may be bounded by a different set of flows which means that the constraints on modifying a bin's size can be different from bin to bin. It is also important to note that the order of the bins is now important since they contain flows belonging to different groups (this was not a problem for $K = 2$ since the bins contained only flows in F_1). For that reason, state independent algorithms are less practical for $K \geq 3$. We therefore concentrate on state dependent algorithms. We further restrict our attention to algorithms that pack the flows in increasing order of their grant intervals, i.e., from F_1 to F_K. The advantage of packing flows in this order is that when the algorithm schedules flows in group F_j it only needs to consider the first interval of length I_j; the allocations in subsequent intervals are then simply duplicated from the first interval.

A state dependent algorithm uses some heuristic that defines in which bin a new flow is to be packed. Examples for heuristics the algorithm may use are Next-Fit, First-Fit, Best-Fit, and Worst-Fit. The algorithm may choose to apply the same heuristic to all the groups, or change the heuristic from group to group. In addition the algorithm must define the rules for changing bin sizes. It can choose, for example, to change a bin's size whenever doing so enables packing the next flow, or only when the flow does not fit in the free space of any other bin. The combination of different packing heuristics with different rules for changing bin sizes defines a large number of possible state dependent algorithms. The algorithm we present here is based on the First-Fit heuristic. It is easy to design similar algorithms based on other heuristics (e.g., Best-Fit) or other rules of changing bin sizes (e.g., pack a flow in the first bin to which it would fit by changing the bin's size).

7.1 Algorithm First-Fit with Jitter for K Intervals - $FFJ - K$

$FFJ - K$ is an offline state dependent approximation algorithm. The algorithm divides the basic interval of length I_K into $m = I_K/I_1$ bins of size I_1. Recall that for a state dependent algorithm we denote by $C(i)$ the free space in bin B_i and by $d(i)$ the displacement (or actual jitter) of the flows in bin B_i. The pair $(C(i),\, d(i))$ describes the state of bin B_i. In the initial state $C(i) = B$

and $d(i) = 0$ for all $1 \leq i \leq m$. Algorithm $FFJ - K$ first constructs a list L in which the flows are sorted according to increasing order of grant intervals; the order among flows with the same grant interval is not important. $FFJ - K$ now goes over all the flows in L and tries to schedule them one by one. The algorithm uses the INSERT procedure (defined in subsection 5.3) to insert a flow into a given bin while maintaining the bins states. The procedure returns SUCCESS if the flow was inserted or FAIL if the flow cannot be inserted (even by increasing the bin's size). When a flow $f(i)$ is inserted into a bin it is allocated the first $S(i)$ free (consecutive) slots in that bin.

Algorithm $FFJ - K$

- Divide the basic interval into I_K/I_1 bins of size I_1.
- Construct a list L in which all the flows are sorted according to increasing order of grant intervals. The order among flows with the same grant interval is not important.
- Go over all the flows in L and try to schedule them one by one. For a given flow f_i with grant interval $I(i)$ and grant size $S(i)$ perform the following steps
  1. Consider the first $n_i = I(i)/I_1$ bins, i.e., the bins in the first interval of length $I(i)$.
  2. If f_i fits in the free space of one of the first n_i bins, insert it in the first bin in which it fits, i.e., select bin B_k such that $k = \min\{k' \leq n_i : \ C(k') \geq S(i)\}$ and execute procedure INSERT($S(i)$, k, n_i).
  3. If the flow does not fit in any of the bins, try to insert it in the first bin *with free space* to which it can be inserted by increasing the bin's size, i.e., insert the flow in a bin B_k such that $k = \min\{k' \leq n_i : \ C(k') > 0$ and INSERT($S(i)$, k, n_i)==SUCCESS$\}$.
  4. If the flow has been inserted in one of the first n_i bins, insert the flow into the bins in all $I_K/I(i)$ subsequent intervals of length $I(i)$, i.e., suppose the flow has been inserted in bin B_k, execute INSERT($S(i)$, $k + j \cdot I(i)/I_1$, n_i) for all $1 \leq j \leq I_K/I(i) - 1$.
  5. If the flow has not been inserted in one of the first n_i bins, discard it.

Note that the algorithm may not be able to insert a given flow f_i in one of the first n_i bins (steps 2 and 3). However, if the flow has been inserted in one of the first n_i bins, the INSERT operations in step 4 cannot fail. The reason is that all subsequent intervals of length $I(i)$ are identical to the first interval into which the flow has been successfully inserted.

The $FFJ - K$ algorithm can be implemented to run in $O(n \cdot m)$ time, where $m = I_K/I_1$ is the number of bins. Assuming $m \ll n$ is fixed, $FFJ - K$ has linear running time.

7.2 Producing perfectly-periodic Schedules

In this subsection we consider the construction of perfectly-periodic schedules; such schedules correspond to the case where $J_{min} = 0$ which means that all the flows must be scheduled exactly in their nominal grant times. We define an algorithm called Perfectly-Periodic First-Fit ($PP{-}FF$) which is identical to algorithm $FFJ - K$ except for step 3 which is removed. We first show that $PP - FF$ indeed produces a perfectly-periodic schedule.

Claim 7.1 *A schedule produced by algorithm $PP - FF$ is perfectly-periodic.*

Proof: To prove the claim we must show that the time between every two grants of any flow f_i is exactly $I(i)$. This property holds if the number of occupied slots is identical in all the bins in which the flow is inserted. Assume $f_i \in F_j$ and the first bin it is inserted into (in step 2) is bin B_k where $1 \leq k \leq I_j/I_1$. The flow is then inserted (step 4) in all other bins $B_{k'}$ satisfying $k' = k + j \cdot I(i)/I_1$, where $1 \leq j \leq I_K/I_j - 1$. Now consider any other flow f_h that has been inserted into bin B_k before f_i. Clearly, $I(i) = mI(h)$, where $m \geq 1$ is an integer. Since f_h is also inserted in any bin $B_{k''} = k + j \cdot I(h)/I_1$, f_h must have been inserted in all the bins f_i has been inserted into. It follows that all the bins f_i is inserted into contain exactly the same set of flows. We conclude that whenever a new flow is scheduled all the bins to which it is inserted have the same number of occupied slots, hence the interval between any two grants is identical. ■

The following theorem establishes an important property of $PP - FF$ in terms of the channel utilization it achieves. We note that a similar result has been obtained by Patt-Shamir et al. in [31], however the algorithm they presented is only suitable for grant intervals which are a power of 2 (times a common constant).

Theorem 5 *Let F be a set of flows with K related grant intervals such that $I_j = m_j\, I_{j-1}$ where m_j is a positive integer for all $2 \leq j \leq K$. Denote by S_{max} the maximal grant size of flows in F, and by $W_F \equiv \sum_{f_i \in F} S(i)/I(i)$ the total bandwidth requirements of flows in F. Algorithm $PP - FF$ always produces a perfectly-periodic schedule for a subset of flows $F' \subseteq F$ such that the channel utilization of the schedule satisfies*

$$\eta = \sum_{f_i \in F'} \frac{S(i)}{I(i)} \geq \min\left\{ W_F,\, 1 - \frac{S_{max} - 1}{I_1} \right\}.$$

Furthermore, no other algorithm can guarantee to produce a perfectly-periodic schedule with a higher channel utilization, i.e., with $\eta > 1 - \frac{S_{max}-1}{I_1}$.

Proof: We show that $PP - FF$ never fails to schedule a flow when the channel utilization is less than $1 - \frac{S_{max}-1}{I_1}$. Assume by contradiction the opposite and denote the first flow the algorithm fails to schedule by f_j. Since the algorithm failed to find a bin with enough free space to fit f_j, it follows that when f_j is scheduled the free space in all the bins is at most $S(j) - 1$. However this means that the channel utilization satisfies $\eta \geq \frac{I_1 - (S(j)-1)}{I_1} \geq 1 - \frac{(S_{max}-1)}{I_1}$, a contradiction. Since $PP - FF$ does not discard any flow before the channel utilization reaches $1 - \frac{(S_{max}-1)}{I_1}$ it can guarantee to achieve the channel utilization stated in the theorem.

We now show that no algorithm can guarantee to produce perfectly-periodic schedules with channel utilization $\eta > 1 - \frac{S_{max}-1}{I_1}$. Consider a set F that has one flow in F_1 with $I(1) = 2S_{max} - 1$ and $S(1) = S_{max}$; in F_2 there are $S_{max} - 1$ flows, the grant interval is $I_2 = S_{max} \cdot I(1)$ and all grant sizes are $S(2) = \ldots = S(S_{max}) = S_{max}$. The total bandwidth requirement is $W_F = 1$. It is easy to verify that in this example there is no way to produce a perfectly-periodic schedule where f_1 is scheduled with one of the flows in F_2, hence the maximum channel utilization an optimal algorithm can achieve is $\eta = \frac{S_{max}}{I(1)} = 1 - \frac{S_{max}-1}{I(1)}$. ■

An immediate corollary of Theorem 5 is that given a set F, algorithm $PP - FF$ can produce a perfectly-periodic schedule of all the flows in F whenever $W_F \leq 1 - \frac{S_{max}-1}{I_1}$. It is important to note that in most cases $PP - FF$ is expected to achieve a higher channel utilization than what Theorem 5 guarantees.

Let us consider the meaning of Theorem 5 for practical applications. Consider a 1 Mbps link that carries VoIP calls. VoIP packets are typically less than 200 Bytes so we expect S_{max} to be about 1.6 msec while I_1 is typically 10 msec. In this example $PP - FF$ can schedule any set of flows (with related grant intervals) achieving channel utilization of $\eta = 0.84$. If the link's speed goes up to 10 Mbps $PP - FF$ achieves almost perfect utilization $\eta = 0.984$.

7.3 Legal Schedule for Bounded Grant Sizes

In subsection 6.3 we analyzed the scheduling problem with two related grant intervals under the assumption that grant sizes are bounded by a constant S_{max}. We presented algorithms that find a legal schedule for any feasible set of flows for which $J_1 \geq S_{max} - 1$. We now derive similar results for the general case of K related grant intervals. We denote by S_{max} the maximal grant size of flows in $F \setminus F_1$, and by J_{min} the minimal grant jitter of flows in $F \setminus F_K$. We show that if $J \geq (K-1)(S_{max} - 1)$ and $W_F \leq 1$ (i.e., the bandwidth requirement of all the flows does not exceed 1) algorithm $FFJ - K$ can always construct a legal schedule of all flows in F.

Theorem 6 *Let F be a set of flows with K related grant intervals such that $I_j = m_j I_{j-1}$ where m_j is a positive integer for all $2 \leq j \leq K$. Denote by S_{max} the maximal grant size of flows in $F \setminus F_1$ and by J_{min} the minimal grant jitter of flows in $F \setminus F_K$. Algorithm $FFJ - K$ produces a legal schedule for all flows in F whenever F satisfies the following two conditions:*

1. $W_F \equiv \sum_{f_i \in F} \frac{S(i)}{I(i)} \leq 1$
2. $J_{min} \geq (K-1)(S_{max} - 1)$

Furthermore, no other algorithm can guarantee to produce a legal schedule of all flows in F, unless F satisfies both conditions 1 and 2.

Proof: The first condition is trivial and only ensures that the total bandwidth requirement does not exceed the capacity of the channel. The jitter requirements are a consequence of the following lemma.

Lemma 1 *Denote by τ_j, $2 \leq j \leq K$, the moment when $FFJ - K$ is about to starts scheduling flows in group F_j. Consider the positions, over the basic interval, of the grants of all the flows that have been scheduled before τ_j and compare them to their positions at time τ_{j+1}, then the difference between the two positions for every flow is at most $S_{max} - 1$.*

Proof: The lemma considers the effects of scheduling a single group F_j of flows; it is therefore sufficient to consider only the first I_j/I_1 bins in the basic interval. Consider the first flow in group F_j that is inserted in step 3 of algorithm $FFJ - K$ and thus causes a displacement of flows in F_1. Denote this flow by f_a and assume it is inserted in bin B_k where $1 \leq k \leq I_j/I_1 - 1$. Since the size of f_a is bounded by S_{max} and the flow is inserted into a bin that has at least one unit of free space, the maximum displacement f_a can cause is $S_{max} - 1$. Inserting f_a into B_k may cause a displacement in the grants to adjacent flows. Suppose the furthest grant from f_a that has been displaced when f_a was inserted is in bin B_l, $k < l \leq I_j/I_1$. We conclude that after f_a is inserted all the bins B_1 to B_{l-1} are full. Therefore, $FFJ - K$ would not try to insert any subsequent flows into these bins. It follows that although some subsequent flow $f_b \in F_j$ may also cause a displacement to flows in F_1, the set of grants which are displaced by f_b is completely different from the set of grants displaced by f_a. To summaries, during the scheduling of flows in F_j by $FFJ - K$ each grant to a flow that has already been scheduled is displaced (from its position at τ_j) at most once and the displacement is by at most $S_{max} - 1$. ■

The proof of the theorem immediately follows from Lemma 1. Each time a new group of flows is scheduled, the displacement of grants to flows in F_1 from their nominal grant times may increase by at most $S_{max} - 1$. There are $K - 1$ such groups ($F_2 \dots F_K$), hence the maximum displacement of flows in F_1 is $(K-1)(S_{max} - 1)$. Since by condition 2 the tolerated jitter J_{min} allows such a displacement, and since by condition 1 $W_F \leq 1$, $FFJ - K$ can schedule all the flows in F.

To see why no other algorithm can guarantee to produce a legal schedule of all flows in F, unless F satisfies both conditions 1 and 2, consider the following example. There are K grant intervals satisfying $I_1 = 2$ and $I_j = (S_{max}+1) \cdot I_{j-1}$, $\forall\, 2 \leq j \leq K$. There is only one flow in each group $F_1, \ldots, F_K$ with the following grant sizes: the size of the flow in F_1 is $S(1) = 1$, the size of all the rest of the flows is $S(2) = \ldots = S(K) = S_{max}$. Figure 6 shows an instance of the example with $K = 3$. It is easy to verify that $W_F \leq 1$ for all values of S_{max} and K. However, there is no way to legally schedule all the flows unless $J \geq (K-1)(S_{max}-1)$. ■

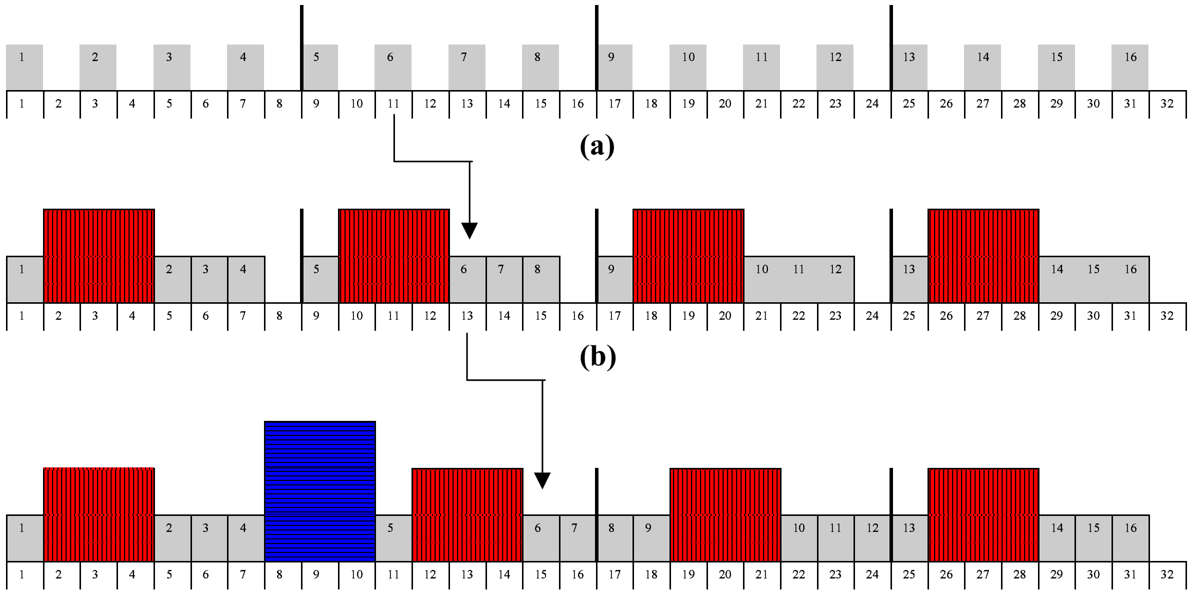

Figure 6: Worst case example of constructing a schedule by algorithm $FFJ-K$ for $S_{max} = 3$ and $K = 3$. Steps (a), (b) and (c) show the schedule after f_1, f_2, and f_3 are scheduled, respectively. The required jitter is $J_{min} \geq 4$; the displacement of grant number 6 of f_1 is indicated by arrows.

Based on the proof of Theorem 6 we can actually give a more precise definition of the jitter requirements of algorithm $FFJ-K$.

Corollary 7.1 *Denote by S_j the maximum grant size in group F_j, and by J_j the minimum grant jitter in group F_j. Algorithm $FFJ-K$ can produce a legal schedule for any set F with $W_F \leq 1$ if and only if the values of grant jitter satisfy*

$$J_j \geq \sum_{n=j+1}^{K} (S_n - 1), \quad \forall\, 1 \leq j \leq K-1. \tag{8}$$

Proof: It follows from Lemma 1 that whenever $FFJ-K$ schedules a new group F_j the displacement of grants of flows in any group that has already been scheduled (i.e., $F_1 \ldots F_{j-1}$) may increase by at most $S_j - 1$. Hence the displacement (actual jitter) of a flow in F_j is never more than the grant jitter specified in (8). The example given in the proof of Theorem 6 shows that the requirements of the corollary are tight (see Figure 6 for an example). ■

7.4 Optimizing the Subset selection

Theorems 5 and 6 specify conditions under which algorithms $PP-FF$ and $FFJ-K$ produce a schedule of all flows in a given set. However, the algorithms may perform poorly when they are applied to sets for which these conditions do not hold. In fact, the worst case performance ratio of the algorithms, as we originally presented them, is not bounded; this fact is demonstrated by the following simple example. Consider the case of two flows withe intervals $I(1) = I_1$ and $I(2) = I_2 = m_2 \cdot I_1$, and sizes $S(1) = 1$ and $S(2) = I_2 - m_2$. Since $PP-FF$ and $FFJ-K$ schedule flows according to increasing grant intervals they only schedule f_1 achieving channel utilization of $\eta = 1/I_1$. An optimal algorithm schedules f_2 achieving channel utilization of $\eta = (I_1 - 1)/I_1$. The performance ratio in this case is $R = I_1 - 1$ which can be arbitrarily large. As we could see from the above example the reason for the poor performance of the algorithms is that they make no attempt to optimize the subset of flows they schedule.

In order to improve the worst case performance ratio of the algorithms a preliminary stage of choosing an optimized subset of flows is required. The problem of choosing an optimal subset $F' \subseteq F$ requiring only that $W_{F'} \leq 1$ is NP-hard (the problem is equivalent to the subset sum problem). We are therefore interested in a simple heuristic that would improve the performance of the algorithm without increasing its complexity. For $PP - FF$ the goal is to select a subset with bandwidth requirements not exceeding unity such that S_{max} is minimized and I_1 (i.e., the shortest grant interval in the subset) is maximized. For $FFJ - K$ the goal is to select a subset of flows with maximum bandwidth requirements such that the conditions in (8) are satisfied.

Denote by W_{F_j} the bandwidth requirement of all flows in group F_j. In order to guarantee that the worst case performance ratio is bounded by K the simple heuristic of scheduling the flows in the group with the highest bandwidth requirement is sufficient. Unfortunately, there is no way to guarantee that any algorithm can achieve a channel utilization $\eta > W_F/K$. This is demonstrated by the following example. Consider K flows such that for $1 \leq j \leq K$, grant intervals are $I(j) = K^j$, sizes are $S(j) = K^{j-1}$, and the tolerated jitter is $J(j) = I(j)$. In this example $W_F = 1$ and the bandwidth requirement of each flow is $W_{f_j} = W_F/K$. It is easy to verify that no two flows can be scheduled together, hence the maximum channel utilization any algorithm can achieve is $\eta = 1/K$.

7.5 Other Scheduling Algorithms

It is possible to define algorithms which are similar to $FFJ-K$ but use different packing heuristics. For example algorithms $BFJ - K$ and $WFJ - K$ would use the best-fit and worst-fit rules, respectively, when selecting the bin in which to insert a new flow. The $FFJ - K$ algorithm can be implemented to run in $O(m \cdot n)$ time (the INSERT procedure requires $O(m)$ operations). If we allow a slightly higher complexity of $O(m \cdot n \log n)$ we may define the First-Fit Decreasing with Jitter ($FFDJ - K$) algorithm. The only difference is that $FFDJ - K$ sorts the flows within each group according to decreasing grant sizes. From a worst case perspective $FFDJ - K$ has the same properties of $FFJ - K$. This is evident since the worst case examples we presented for $FFJ - K$ also hold for $FFDJ - K$. However, we expect $FFDJ - K$ to perform better than $FFJ - K$ on average since sorting is known to help in bin packing problems.

The NFJ algorithm, which we presented in subsection 6.2.1, has $O(n)$ running time and the same worst case performance guarantees as $FFJ - K$ for $K = 2$. However, extending NFJ to run in (strict) linear time for $K \geq 3$, while maintaining the same properties of $FFJ - K$, is not straightforward. The problem is that the algorithm must know which bins contain free space whenever it starts packing a new group. To maintain linear running time the algorithm should therefore use some kind of indexing information to indicate which bin should be the next bin to be opened.

8 Unrelated Grant Intervals

So far we considered only the case of related grant intervals. Although we expect to find related grant intervals in many applications, in the general case the different grant intervals are not necessarily integer multiples of each other. In this case the scheduling problem becomes more complex. The higher complexity can be demonstrated by considering the case of unit grant sizes and zero jitter (i.e., $S(i) = 1, J(i) = 0, \forall i$). This problem is easily solved if grant sizes are related (for example by using $PP - FF$) but is NP-hard in the case of unrelated grant intervals [27].

From our point of view the main problem is that for unrelated grant intervals it is impractical to model the scheduling problem as a bin packing problem since the different grant intervals do not overlap. As a result, the algorithms we presented previously cannot be applied to a set of unrelated grant intervals.

We present two approaches for handling unrelated grant intervals. The first approach is to round the requested set of unrelated grant intervals to a set of related grant intervals. The second approach is to maintain the original grant intervals and try to schedule them the best we can.

8.1 Rounding of Unrelated Grant Intervals

The rounding process is used to transform a set of unrelated grant intervals to a set of related grant intervals. The motivation for rounding is to keep the scheduling problem simpler and to be able to use the algorithms we developed for related grant intervals; these algorithms are more efficient and of lower complexity compared to algorithms for unrelated grant intervals. For a given flow f_i with parameters $S(i),\ I(i),\ J(i)$ we denote the parameters of the rounded flow $\hat{f}_i$ by $\hat{S}(i),\ \hat{I}(i),\ \hat{J}(i)$. In order to change the original grant intervals by as little as possible we round them to be powers of 2 (or powers of 2 times some constant).

The rounding rules depend on the application. Ideally the parameters of a new flow should be negotiated (between the application and the scheduler) during flow setup. This way the application is informed of the available values for grant intervals and may choose the grant size and tolerated grant jitter accordingly. Unfortunately, some applications may not be able to perform this kind of negotiation. In such cases the scheduler must determine the new set of parameters based on the requested parameters and the type of application (in case it is known). We distinguish between two kinds of applications.

1. Applications with fixed packet size - These applications expect to receive packets of specific format and of fixed size. As a result the scheduler has no flexibility in changing the requested grant size and must therefore set $\hat{S}(i) = S(i)$. As a typical example consider applications, such as broadcast disks [26], which are used to deliver messages to clients in periodic intervals.

2. Applications with flexible packet size - These applications can handle packets of different sizes. As a result the scheduler may modify the grant size when modifying the grant interval. Packets usually carry a fixed size component (such as headers) which is duplicated in every packet regardless of its size. In order to maintain the application's requested data rate, when the scheduler decides to modify the grant size it must take into account packets format. Application with flexible packet size include most applications that deliver multimedia (voice, video or data) streams over higher layer protocols such as TCP or UDP.

Example: Consider a VoIP application with a voice codec that produces a CBR output data stream of 64 kbps. The data is packetized at 15 ms intervals, thus producing 120 bytes of payload each 15 ms. Each voice packet must also carry the RTP, UDP, and IP protocol headers which account for an extra 40 bytes; we ignore any other overheads, such as the link layer overhead, in this example. Assume each slot is 0.1 ms and carries 10 bytes. The parameters the application is likely to request for such a flow f_i are $S(i) = 16$, $I(i) = 150$. Suppose the scheduler decides to use only intervals of 5, 10, 20, and 40 ms; it therefore rounds the requested grant interval to 10 ms, i.e., $\hat{I}(i) = 100$. The value of $\hat{S}(i)$ depends on the type of the application. A fixed packet size application would use $\hat{S}(i) = S(i) = 16$. If the application supports flexible packet sizes the scheduler should maintain a data rate of 64 kbps. It can therefore use 120 bytes packets (80 bytes payload + 40 bytes headers) every 10 ms, hence $\hat{S}(i) = 12$. In this example the ability to change the grant size provided a 25% reduction in the bandwidth requirement of the rounded flow.

We now analyze the effects of rounding on the bandwidth requirements of the flows for the two types of applications we mentioned. The performance of the scheduling algorithms are then easily deduced. Assume the bandwidth requirements of all flows increase by a factor of x due to rounding, i.e., $\hat{W}_F = xW_F$. Let A be an algorithm designed for related grant intervals. Suppose A guarantees that its channel utilization is at least η when it is applied to rounded sets. When A is used to schedule unrelated sets the channel utilization it achieves is at least η/x.

8.1.1 Applications with Fixed Packet Size

We distinguish between two cases. If the application requests fixed size packets but has no bandwidth requirements, we may round up the grant interval to the power of 2 if the tolerated jitter allows it. If the application has bandwidth requirements, we always round the grant interval down to the nearest power of 2. The reason is that, even if the jitter allows rounding to the nearest higher power of 2, we would still have to use $\hat{S}(i) = 2S(i)$ in order to satisfy the required data rate. For applications with no bandwidth requirements rounding is performed as follows (logarithms are base 2)

$$\hat{I}(i) = 2^{\lfloor \log(I(i)+J(i)) \rfloor}\,; \quad \hat{S}(i) = S(i)\,; \quad \hat{J}(i) = J - \max\{\hat{I}(i) - I(i),\, 0\} \tag{9}$$

For applications with bandwidth requirements rounding is to the lower power of 2

$$\hat{I}(i) = 2^{\lfloor \log I(i) \rfloor}\,; \quad \hat{S}(i) = S(i)\,; \quad \hat{J}(i) = J \tag{10}$$

Rounding in the case of fixed size packets increases (or decrease) the bandwidth requirements of a flows by a factor of $I(i)/\hat{I}(i)$. In the worst case the increase is by a factor of $2 - \frac{1}{\hat{I}(i)}$. Assuming uniform distribution of requested intervals in the range $[\hat{I}(i)+1,\, 2\hat{I}(i)]$ (and zero jitter) the average increase is by a factor of 1.5. The above results mean that the performance of an algorithm designed for related flows may drop by at most 50% when the algorithm is used for scheduling unrelated flows. If grant intervals are uniformly distributed the performance drops by 33%. In practice we can expect an even smaller decrease in performance since a large portion of the intervals would not need rounding.

8.1.2 Applications with Flexible Packet Size

In the case of flexible packet size we choose to round the grant interval to the nearest lower or upper power of 2 depending on the tolerated jitter. Rounding to a higher grant interval is desirable as it results in less overhead. We assume each packet contains h slots of overhead which the requested grant size already includes. Rounding is performed as follows.

$$\hat{I}(i) = 2^{\lfloor \log(I(i)+J(i)) \rfloor} \; ; \quad \hat{S}(i) = \left\lceil \frac{(S(i)-h)\hat{I}(i)}{I(i)} + h \right\rceil \; ; \quad \hat{J}(i) = J - \max\{\hat{I}(i) - I(i),\, 0\} \tag{11}$$

In the case of flexible packet size rounding the grant interval toward the upper power of 2 will (in most cases) reduce the bandwidth requirements of a flow. If rounding is toward the lower power of 2 the bandwidth requirements of the flow increases. The factor by which the bandwidth requirements increase is

$$\frac{\hat{S(i)}/\hat{I}(i)}{S(i)/I(i)} \le \frac{(S(i)-h)\hat{I} + (h+1)I(i)}{S(i)\hat{I}(i)} \le 1 + \frac{h+2}{S(i)}$$

As we can see the increase in the bandwidth requirements depends on the ratio $h/S(i)$. For multimedia applications running over IP this ratio is typically around 0.3 but drops to about 0.1 when header suppression is used. Assuming $h/S(i) = 0.1$ the performance of an algorithm designed for related intervals may drop by at most 9% when it is used for scheduling flows with unrelated intervals. The above result indicates that rounding is a very good option for applications with flexible packet size.

8.1.3 Buffer Considerations

When rounding the grant interval of a flow the delay between every two packets is bound to change. In order to compensate for this change some kind of packet buffering is required at both the sending and receiving sides. Fortunately, if we follow the rounding rules we presented earlier the buffering requirements are minimal. In particular, since the difference between the requested and rounded grant intervals is by at most a factor of two, a buffer capable of holding two packets of the original size is enough. In addition the receiving side should delay the delivery of the first packet to the application by one grant interval. Since buffers are maintained by almost all application the buffer requirements imposed due to rounding are insignificant. Figure 7 present an example of a voice application with a CODEC that produces packets of 3 slots every 15 ms and thus requests $I(i) = 15$, $S(i) = 3$. The scheduler rounds the application's request to $\hat{I}(i) = 10$, $\hat{S}(i) = 2$. We can see that in this case a buffer of $S(i)$ at both sides is sufficient. The receiver delays the delivery of the first packet to the application by $I(i)$; after this initial delay the application is delivered packets according to the original request of the application, i.e., a packet of size $S(i) = 3$ every $I(i) = 15$ ms.

8.2 Algorithms for Scheduling Unrelated Grant Intervals

We now explore the possibility of scheduling flows with unrelated grant intervals without rounding. In order to produce a legal schedule we must now find a legal allocation of the flows over a basic interval whose length equals the least common denominator (LCD) of all grant intervals. This basic interval can then be repeated to produce a legal schedule.

When designing an algorithm for scheduling unrelated grant intervals we cannot use a bin packing algorithm and therefore a new approach must be taken. One possible solution is to schedule the flows one by one, deciding for each flow whether or not it can be scheduled. To

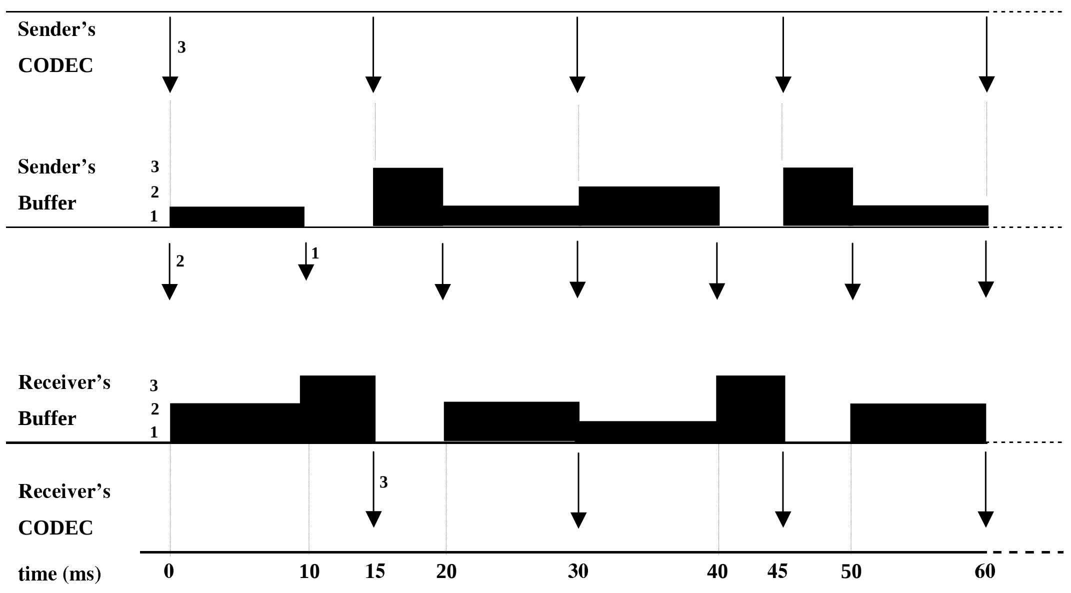

Figure 7: Example of buffers content when the scheduler rounds the application's request for $I(i) = 15$, $S(i) = 3$ to $\hat{I}(i) = 10$, $\hat{S}(i) = 2$. A buffer of $S(i)$ at both sides is sufficient in this case.

do so we construct an array of length I, where I equals the least common denominator of all grant intervals of the flows that we want to schedule. Each array cell represents a slot and can be assigned to at most one flow.

We now describe a general framework for a scheduling algorithm that uses an array of allocations. Basically the scheduling algorithm decides in which order to schedule the flows and in what way to allocate each flow in the array. When a new flow f_k is scheduled the algorithm performs the following operations:

1. Select a time reference $t_0(k)$ for the flow. The time reference may take any value $1 \leq t_0(k) \leq I$.

2. Set allocation number $n = 0$.

3. Try allocating $S(k)$ consecutive array cells to f_k, starting from any cell in the range $t_0(k) + n\,I(k)$ to $t_0(k) + n\,I(k) + J(k)$, all operations are module I. At this stage it is possible to change the allocations of other flows provided that their timing requirements are maintained.

4. If the allocation succeeded set $n = n + 1$.
   Otherwise discard f_k, clear all allocation that were already made for f_k, and move on to handle the next flow.

5. If $n = I/I(j)$ then f_k has been successfully scheduled.
   Otherwise go back to step 3.

The above description provides guidelines for constructing a scheduling algorithm. We now present one such algorithm in more details.

Algorithm Shortest Interval First

The algorithm schedules the flows according to increasing order of their grant intervals $I(k)$. It tries to schedule each flow f_k performing the following steps:

1. Scan the array from cell one and select a time reference $t_0(k)$ in the first position where there are $S(k)$ free cells.
2. Set allocation number $n = 0$.
3. If there are $S(k)$ consecutive free array cells in the range $t_0(k)+n\,I(k)$ to $t_0(k)+n\,I(k)+J(k)$ (operations are module I), allocate $S(k)$ consecutive array cells to f_k starting from the lower index cell possible.
   Otherwise try to create a sequence of $S(k)$ consecutive free array cells by moving some of the flows according to their tolerated jitter.
4. If the allocation was successful set $n = n + 1$.
   Otherwise discard f_k and restore the array to its structure before allocating f_k.
5. If $n = I/I(j)$ then flow f_k has been successfully scheduled.
   Otherwise go back to step 3.

The simplest algorithm can ignore the jitter of all other flows and use the jitter of the current flow only. A more complex algorithm can try to use the jitter of other flows. In this case we must save more information on each allocation. If we find out that we can allocate the flow in the current array we accept the flow, otherwise the flow is rejected. When a flow is removed we delete its allocations from the array.

9 Online Scheduling Algorithms

In the online version of the scheduling problem flows arrive (established) one after another and the algorithm must either schedule an arriving flow or reject it. We assume the algorithm has no knowledge about the arrival of future flows. The objective of the scheduling algorithm is to maximize the channel utilization. We distinguish between two types of flows depending on their duration: *permanent flows* that have an infinite duration, and *temporary flows* that may terminate over time. We analyze the problem assuming permanent flows and then explain how the results change in the case of temporary flows. The algorithms we develop are suitable for both permanent and temporary flows.

We first show that in the online problem the performance ratio of any deterministic algorithm is not bounded. Let us consider a simple example where all the flows have the same grant interval, I_1. The algorithm is first given flows of size 1, if the algorithm accepts a flow the next flow to arrive is of size I_1. The performance ratio in this example is I_1. To avoid this problem one may suggest the following rule: an algorithm must not reject a flow if it can be accepted. As this rule applies to both our algorithm and the optimal algorithm, it solves the problem of online scheduling over a single interval. Unfortunately, the above rule is not enough even in the case of two related grant intervals, as we demonstrate in the following claim.

Claim 9.1 *For two grant intervals $I_2 = m_2 \cdot I_1$ and zero tolerated jitter, the worst case performance ratio of any deterministic online algorithm is at least $R_A \geq I_1$.*

Proof: To show that the performance ratio of any deterministic algorithm cannot be less than I_1, consider the following example. The first m_2 flows that arrive are in F_2 and have a size of one. If the algorithm allocates all these flows in consecutive slots we choose the next flow to be in F_1 and of size $I_1 - 1$. The algorithm must reject the flow hence its utilization is $\frac{m_2}{I_2}$. An optimal algorithm can accept all the flows and achieve a utilization of 1. The performance ratio in this case is $R_A = \frac{I_2}{m_2} = I_1$. If the algorithm allocates the first m_2 flows in non consecutive slots we choose a flow in F_2 and of size $I_2 - m_2$. The flow must be rejected by the algorithm and the performance ratio is again $R_A = I_1$. ■

The reason for the unbounded performance ratio in the above examples is the large grant sizes of the flows which are rejected by the algorithm. In fact, if we enforce the rule that an online algorithm must not reject a flow if it can be accepted, an optimal online algorithm may also perform poorly. As an example consider a problem with zero tolerated jitter where the first flow to arrive is in F_K and its grant size is larger than $I_1/2$. The next two flows are in F_1 and have grant size of $I_1/2$ each; these flows must be rejected by any algorithm. In this example the ratio between the performance of the optimal online algorithm and an optimal offline algorithm is $2I_K/I_1$.

In order to provide meaningful results we study the online problem under the (realistic) assumption that grant sizes are bounded and are typically much smaller than any grant interval. We assume there are K related grant intervals $I_1 < I_2 < \ldots < I_K$, that is, $I_j = m_j \cdot I_{j-1}$, for every $2 \leq j < K$, where m_j is a positive integer. We denote by S_j the maximum grant size of all flows with grant interval I_j. We define $S_{max} = \max\{S_j\}$ and assume that $S_{max} < I_1$. Similarly J_j denotes the minimal grant jitter of all flows with grant interval I_j and $J_{min} = \min\{J_j\}$.

The tolerated jitter plays an important role in the design of an online scheduling algorithm. The scheduling algorithms we develop become more efficient the larger the tolerated jitter is. We identify three categories of tolerated jitter: large, small, and zero jitter. The algorithms for each category along with their performance analysis are presented in the following subsections.

9.1 Online Algorithms for Large Tolerated Jitter

The category of large tolerated jitter includes problems where the flows to be scheduled satisfy the following

$$J_j \geq I_j\,, \quad 2 \leq j \leq K \tag{12}$$

The advantage of a large tolerated jitter is that it enables the use of offline algorithms for the online scheduling problem. Whenever a new flow is established we let the offline algorithm construct a new schedule, and start using the new schedule at the next basic interval. The new schedule is not accepted if accepting a new flow results in rejecting a previously accepted flow. The main obstacle we face is that during the transition from one offline schedule to another, the jitter of the flows may increase. The online algorithm should therefore check that a new schedule does not violate the tolerated jitter of any admitted flow.

Let A be an offline algorithm for scheduling flows with related grant intervals. We construct an algorithm OA, which is an online version of A, in the following way. Denote by F_c the set of flows that OA is currently scheduling; before the schedule begins $F_c = \emptyset$. Algorithm OA operates as follows

- When a new flow f_i is established
  1. Use the offline algorithm A to construct a schedule for the flows in $F_c + f_i$.
  2. Discard flow f_i if one of the following holds for the new schedule.
     - Not all the flows have been scheduled.
     - One of the flows becomes jittered by more than its tolerated jitter.
  3. Otherwise, set $F_c = F_c + f_i$ and start using the new schedule at the beginning of the next basic interval.
- When a flow f_i ends, set $F_c = F_c - f_i$.

9.1.1 Algorithm Online Perfectly-Periodic First-Fit

We now choose algorithm $PP - FF$ (introduced in Subsection 7.2) as an offline algorithm and explore the jitter and performance guarantees of the Online Perfectly-Periodic First-Fit ($OPP - FF$) algorithm. Recall that $PP - FF$ schedules flows according to increasing order of grant intervals and is able to maintain zero jitter. The jitter requirements of $OPP - FF$ are only due to the fact that it should be able to schedule flows in any given order.

Theorem 7 *Let F be a set of flows with K related grant intervals such that $J_1 \geq 0$ and $J_j \geq I_j$, $2 \leq j \leq K$. Assume that algorithm $OPP - FF$ is used to online schedule flows in F. Denote by $F(t)$ the set of flows that have arrived until time t, and by $\eta(t)$ the channel utilization of the algorithm at time t. The schedule produced by $OPP - FF$ satisfies*

$$\eta(t) \geq \min\left\{W_{F(t)},\, 1 - \frac{S_{max} - 1}{I_1}\right\}.$$

Proof: According to Theorem 5 the offline algorithm ($PP - FF$) can schedule any subset of flows in F achieving the above channel utilization. To prove the theorem it remains to show that the jitter requirements of all the flows are maintained. Let us first consider a flow in F_j, $j \geq 2$. To schedule the flow $PP - FF$ must insert it in one of the bins in the first I_j interval. If the time reference of the flow is fixed to the beginning of the basic interval, the jitter of the first grant could not be more than I_j. Since subsequent grants are exactly I_j apart their jitter is the same.

Now consider a flow in F_1. Here zero jitter is maintained since flows in other groups do not effect the locations of grants to flows in F_1 ($PP - FF$ does not change bin sizes). To maintain zero jitter it is therefore sufficient for the algorithm to keep the order of the flows in F_1 according to the order they arrived (this rule is also useful for other groups). ■

Algorithm $OPP - FF$ has good performance when the tolerated jitter of the flows satisfies $J_j \geq I_j$. When the tolerated jitter decreases the performance of the algorithm is expected to degrade. For example if the first flow has $I(1) = I_j$ and $J(1) = 0$, it prevents the algorithm from accepting new flows with grant interval smaller than I_j. There are ways to improve the performance of the algorithm for problems in which $J_j < I_j$. One immediate improvement is to allocate flows in F_1 at the end (i.e., the last free slots) of each bin; this way both flows in F_1 and F_2 are not jittered. As we mentioned, accepting a new flow with grant interval I_j may only increase the jitter of flows with grant interval larger than I_j. To reduce this jitter we may artificially insert flows from each group into F_c. The artificial flows reserve slots for flows that are expected to arrive in the future. When a new flow arrives it may simply take the place of an artificial flow in its group and hence cause no jitter to other flows.

A drawback of algorithm $OPP - FF$ is its running-time complexity. Whenever a new flow arrives $PP - FF$ is invoked so the complexity of scheduling each new flow is $O(n \cdot m)$, where n is the number of admitted flows and m is the number of bins. Note that in practice, when a flow in F_j arrives $PP - FF$ should only reschedule flows with grant interval larger than I_j; the positions of all other flows is not affected by the new flow.

9.2 Online Algorithms for Small Tolerated Jitter

We say that a set of flows belongs to the category of small tolerated jitter if $J_j < I_j$. The small tolerated jitter may render online versions of offline algorithms inefficient. We therefore develop a new algorithm, called Online Least-Loaded (OLL), for the problem. The main difference between OLL and the offline algorithms we presented earlier is that OLL schedules the flows according to the order they arrive, whereas the offline algorithms schedule flows according to increasing grant intervals.

9.2.1 Algorithm Online Least-loaded - OLL

OLL is based on the least loaded rule, that is, the algorithm prefers to insert a flow into the bin with the least number of occupied slots. We assume that the algorithm knows the value of I_1 before scheduling begins. We also assume that the algorithm knows the size of I_K (the largest interval); this assumption, however, is not essential as the value of I_K can change dynamically during the schedule. As in our previously described algorithms, OLL schedules flows over the basic interval I_K; the basic interval is repeated over and over to define a legal schedule. The basic interval is divided into bins of length I_1; bins are numbered 1 to I_K/I_1. We define the level of bin B_k to be the number of occupied slots in bin number k and denote it by $l(B_k)$. When OLL inserts a flow f_i into a bin the flow is allocated the first free $S(i)$ slots in the bin. The exception to this rule is when $f_i \in F_1$ in which case the flow is allocated at the end of the bin, i.e., in the last $S(i)$ free slots; as will be shown, this ensures that flows in F_1 are not jittered. OLL schedules the flows according to the order they arrive. When a new flow f_i arrives the algorithm performs the following steps.

Algorithm *OLL*

1. Choose the least loaded bin among the bins in the first $I(i)$ interval (ties broken in favor of lower index). Assume the chosen bin is B_k.
2. If f_i can be scheduled starting from B_k, that is, the level of every bin $B_{k'}$ where $k' = k + nI(i)/I_1$, $(n = 0, \ldots, I_K/I(i) - 1)$, satisfies $l(B_{k'}) + S(i) \leq I_1$, then insert f_i into every bin $B_{k'}$.
3. Otherwise (the flow could not be inserted into one of the bins) discard f_i.

OLL requires $O(m)$ operations to schedule each flow, where $m = I_K/I_1$ is the number of bins. In this respect it has an advantage over online algorithms which are based on offline algorithms, such as $OPP-FF$, that require $O(n \cdot m)$ operations per flows, where n is the number of admitted flows. We now analyze the performance of OLL in terms of the channel utilization it achieves.

9.2.2 Performance Analysis of Algorithm *OLL*

To simplify the presentation we analyze the performance of algorithm OLL when it is applied to sets with K related grant intervals which are powers of 2 times I_1, i.e., $I_j = 2^{j-1}I_1$ for every $2 \leq j \leq K$. The analysis for general related grant intervals is almost identical and yields the same final result. The following theorem provides guarantees on the channel utilization that OLL achieves.

Theorem 8 *Let F be a set of flows with K related grant intervals which are powers of 2 times I_1. Denote by S_{max} the maximum grant size in F and require that grant jitters in F satisfy $J_1 \geq 0$, $J_j \geq \min\{I_1, (K-1)S_{max}, (2^{K-j}-1)S_{max}\}$, $\forall j \geq 2$. When algorithm Online Least Loaded (OLL) is used to online schedule the set F the channel utilization achieved satisfies*

$$\eta \geq \min\left\{W_F,\, 1 - \frac{KS_{max} - 1}{I_1} + \frac{K(K-1)S_{max}}{2I_K}\right\} \tag{13}$$

Proof: We first prove two claims that state important properties of algorithm OLL.

Claim 9.2 *Let B_k be any bin in the first I_j interval $(1 \leq j \leq K-1)$, i.e, $1 \leq k \leq 2^{j-1}$ and let $B_{k'}$ be the corresponding bin in the following I_j interval, i.e., $k' = k + 2^{j-1}$. The levels of bins B_k and $B_{k'}$ at every stage of the schedule by OLL satisfy $1 - S_{max} \leq l(B_k) - l(B_{k'}) \leq S_{max}$.*

Proof: We first prove that $l(B_k) - l(B_{k'}) \leq S_{max}$. The proof is by induction on the number of flows OLL scheduled. The property is valid before the first flow is scheduled since both bins are empty. Assume it remains valid after f_{i-1} is scheduled and consider the next flow f_i. We distinguish between two cases depending on whether $I(i)$ is smaller or larger than I_j; we show that in both cases after scheduling f_i the level of B_k cannot exceed the level of $B_{k'}$ by more than S_{max}.

1. $I(i) \leq I_j$ - In this case, if f_i is inserted into B_k it is also inserted into $B_{k'}$. Hence, there is no change in the difference between the levels of B_k and $B_{k'}$.
2. $I(i) > I_j$ - In this case OLL chooses the least loaded bin from an interval that includes both B_k and $B_{k'}$. By the induction assumption we know that before f_i is scheduled $l(B_k) - l(B_{k'}) \leq S_{max}$. Clearly the property may be broken only if OLL decides to insert f_i into B_k. However, OLL would make this decision only if the levels of B_k and $B_{k'}$ satisfy $l(B_k) \leq l(B_{k'})$. Hence, after f_i is scheduled we have $l(B_k) \leq l(B_{k'}) + S(i)$ and, since $S(i) \leq S_{max}$, $l(B_k) - l(B_{k'}) \leq S_{max}$.

The proof that $l(B_{k'}) - l(B_k) \leq S_{max} - 1$ is similar. Flows with grant interval less or equal I_j do not create a difference between the levels of B_k and $B_{k'}$. A flow with grant interval larger than I_j is inserted into $B_{k'}$ only if $l(B_{k'}) < l(B_k)$ which means that after scheduling f_i the levels of the bins must satisfy $l(B_{k'}) < l(B_k) + S(i) \leq l(B_k) + S_{max}$. ■

Claim 9.3 *Let B_h^k and B_l^k be the bins with the highest and lowest levels in the first interval of length I_k ($2 \leq k \leq K$), respectively. Throughout the schedule, algorithm OLL maintains the levels of the bins such that $l(B_h^k) - l(B_l^k) \leq (k-1)S_{max}$.*

Proof: The proof is by induction on k (recall that in an interval I_k there are 2^{k-1} bins).
Base $k = 2$ - The interval I_2 contains only two bins. Flows with grant interval I_1 are inserted into both bins and hence do not create a difference in their levels. When OLL schedules a flows with grant interval $I_j \geq I_2$ it considers the levels of both bins in I_2; the algorithm inserts the flow into the bin with the lower level. As a result the difference that flows with grant interval $I_j \geq I_2$ may create in the levels of B_h^2 and B_l^2 is at most S_{max}.
Step - Assume the claim holds for $k-1$, that is, $l(B_h^{k-1}) - l(B_h^{k-1}) \leq (k-2)S_{max}$. We must show that $l(B_h^k) - l(B_l^k) \leq (k-1)S_{max}$.

Let us first consider the contribution of flows with grant interval $I_j \geq I_k$ to the difference $l(B_h^k) - l(B_l^k)$. A flow in this category is inserted by OLL into the least loaded bin among bins in I_j. Hence, if the flow is inserted into one of the bins in the first I_k interval, this bin is necessarily B_l^k. It follows that after inserting a flow in this category either the difference $l(B_h^k) - l(B_l^k)$ decreases, or (if it increases) is at most S_{max}.

Now consider the contribution of flows with grant intervals smaller than I_k. A flow with grant interval $I_j \leq I_{k-1}$ must be inserted in one of the bins in the first interval of length I_{k-1}. Furthermore, when a flow is inserted in bin number x ($1 \leq x \leq 2^{k-2}$) in the first interval of length I_{k-1}, it is also inserted in bin number $y = x + I_{k-1}/I_1$ in the following interval of length I_{k-1}. Hence, bins number x and y contain exactly the same set of flows with grant intervals $I_j \leq I_{k-1}$. It follows that the contribution of flows with grant intervals smaller than I_k to the difference $l(B_h^k) - l(B_l^k)$ cannot exceed $l(B_h^{k-1}) - l(B_l^{k-1})$. Using the induction assumption we assert that the difference in the levels of B_h^k and B_l^k due to flows with grant interval $I_j < I_k$ is at most $(k-2)S_{max}$.

To summarize, all the flows with grant interval $I_j \geq I_k$ combined may contribute at most S_{max} to the difference in the levels of B_h^k and B_l^k while flows with grant interval $I_j < I_k$ may add another $(k-2)S_{max}$ to this difference. Therefore, the total difference in the levels of B_h^k and B_l^k cannot exceed $(k-1)S_{max}$. ■

We now go back to proving the theorem. Note that when the first flow is rejected by OLL the level of at least one of the bins must be larger than $I_1 - S_{max}$ (otherwise the flow can be inserted in all bins and is not rejected). According to Claim 9.3 the levels of of all other bins at this point is more than $I_1 - KS_{max}$. We conclude that, when the first flow is rejected, the total number of occupied slots in the entire I_K interval, which we denote by C_K, must satisfy

$$C_K \geq I_K - \frac{I_K(KS_{max} - 1)}{I_1} + (K-2)S_{max} \tag{14}$$

We can use Claim 9.2 to get a better lower bound on C_K. According to the claim the difference between the levels of two corresponding bins cannot be more than S_{max} (recall that two bins corresponds when one bin is the kth bin in the first I_j interval and the other bin is the kth bin in the subsequent I_j interval). We know that the level of one of the bins is larger than $I_1 - S_{max}$; denote this bin by B_x. The level of the bin that corresponds to B_x must be more than $I_1 - 2S_{max}$.

The level of the bin that corresponds to that bin must be more than $I_1 - 3S_{max}$, and so on until the level of the K^{th} corresponding bin must be more than $I_1 - KS_{max}$. We can therefore update the lower bound on C_K to be

$$\begin{aligned} C_K &\geq \sum_{n=1}^{K}(I_1 - nS_{max} + 1) + (\frac{I_K}{I_1} - K)(I_1 - KS_{max} + 1) \\ &= \frac{I_K}{I_1}(I_1 - KS_{max} + 1) + \frac{K(K-1)S_{max}}{2} \end{aligned} \tag{15}$$

The total number of slots is I_K, hence the channel utilization when the first flow is rejected is at least what the theorem guarantees

$$\eta = \frac{C_K}{I_K} \geq 1 - \frac{KS_{max} - 1}{I_1} + \frac{K(K-1)S_{max}}{2I_K} \tag{16}$$

Let us now consider the jitter requirements of OLL. Note that the jitter caused by OLL is only due to the different levels of the bins that a flow is inserted into. This difference is clearly no more than the bin size I_1; furthermore, Claim 9.3 tells us that it is not more than $(K-1)S_{max}$. For flows with large grant intervals the required jitter is even lower. A flow with grant interval I_j is inserted into 2^{K-j} corresponding bins. According to Claim 9.2 the difference in the levels of these bins is no more than $(2^{K-j} - 1)S_{max}$. For example, flows in F_K are scheduled without jitter. OLL also maintains zero jitter for flows in F_1 by allocating a new flow in F_1 at the end of each bin, just before previously inserted flows in F_1. Since each flow in F_1 is inserted in all the bins, the number of slots occupied by flows in F_1 is equal in all bins, hence a flow in F_1 is allocated in the same position in all the bins and is thus not jittered. ■

Note that Claims 9.2 and 9.3 remain valid for general related grant intervals, i.e., when $I_j = m_j I_{j-1}$. As a result, Theorem 8 holds for any set of related grant intervals. The claims are based on the property that flows with grant interval I_j do not create a difference of more than S_{max} in the levels of bins in the I_j interval. This property does not depend on the relations between grant intervals. The only modification we need in the proof of the theorem is that in Claim 9.2 the level of a bin in the first I_j interval should now be compared to the levels of the corresponding bins in the next $m_{j+1} - 1$ intervals of length I_j.

We point out that the lower bound of Theorem 8 on the channel utilization achieved by OLL is tight only in certain cases. In order for the bound to be tight (or almost tight) we must have either $I_1 \approx KS_{max} \gg 1$, or $I_1 \gg KS_{max}$. In other cases the worst case channel utilization is higher and depends on the exact values of I_1, K, and S_{max}. As a worst case example for which the bound is (almost) tight consider $K = 3$ and $I_1 = 3S_{max}$. First we have five flows in F_3 with the following grant sizes: 2, 2, 1, 1, and S_{max}; next a flow in F_2 with size S_{max} arrives. The levels of bins B_1, B_2, B_3, and B_4 are now $S_{max} + 2$, 2, $2S_{max} + 1$, and 1, respectively. A new flow in F_1 with grant size S_{max} is now rejected. The channel utilization achieved by OLL in the example is $\eta = 0.25 + 1/2S_{max}$ while the bound of Theorem 8 is $\eta \geq 0.25 + 1/3S_{max}$. We also point out that in this example an optimal algorithm can accept two additional flows in F_1 with grant size S_{max} achieving utilization of 0.92. This indicates that a good estimation to the worst case performance ratio of OLL can be obtained simply by taking the inverse of the lower bound on the channel utilization in Theorem 8 (the performance ratio is always lower than this estimation).

9.3 Online Algorithms for Zero Tolerated Jitter

Zero tolerated jitter means that the schedule should be perfectly-periodic. One way to achieve perfect periodicity is to reserve fixed locations for the flows in each group. A simple (yet not efficient) way to do so is to assign a fixed number of slots to each group in every bin (interval of length I_1). When a new flow is established the algorithm tries to schedule it in the slots that have been assigned to the flow's group. More sophisticated algorithms may allow the reservations to change dynamically. This way, when the locations reserved for a certain group have all been used, the algorithm assigns new slots to that group (from a shared pool or on the expense of a different group). Using fixed reservations algorithms may work well if the expected number of flows and the flows' parameters are known to the algorithm in advance. However, such algorithms may perform poorly if the actual set of flows is considerably different from the expected set of flows.

We use a different approach and slightly modify algorithm OLL in order to produce perfectly-periodic schedules. We call the modified algorithm Perfectly-Periodic OLL ($PP - OLL$). Recall that OLL groups all allocation in a bin into a consecutive block at the beginning of the bin; when a flow is inserted into a bin its grant size is allocated at the end of this block. Since the levels of the different bins into which the same flow is inserted can be different, the flow may be jittered. To avoid this jitter $PP - OLL$ schedules a flow f_i in the following way. The bins into which to insert the flow are chosen exactly as in OLL. After the bins are selected the algorithm scans them to learn what is the highest position of an occupied slot in these bins (slots allocated for flows in F_1 are ignored in this process). Assume the highest position was found to be p. If there are $S(i)$ free slots between slot $p + 1$ and the beginning of slots occupied by flows in F_1 the flow is accepted, otherwise it is rejected. If f_i is accepted it is allocated $S(i)$ (free) consecutive slots starting from position $p + 1$ in each bin f_i should be inserted into. Since every flow that is scheduled by $PP - OLL$ is placed in exactly the same position in all the bins that the flow occupies, the flow is not jittered.

We chose to present $PP - OLL$ in a very simple form. A more sophisticated version of the algorithm would search for other position into which a flow can be inserted and not give up after the first attempt. The best option in terms of performance is to go from the first slot to the last slot and consider every one of them as a starting position for inserting the flow. Unfortunately, even in its most sophisticated version algorithm $PP - OLL$ has a poor worst case performance. We demonstrate this fact with the following example. There are K related grant intervals which are a power of 2 times I_1. We choose $I_1 = nS_{max}$ and $S_{max} \gg 1$. The flows in the example are either from F_K or F_2 and they arrive in n iterations. In iteration number i two flows in F_K with grant size $S_{max} - n + i$ arrive and are followed by two flows in F_2 with grant size 1. Figure 8 presents the content of the first four bins after the first and second iteration in an example where $S_{max} = 4$ and $n = 3$. Note that a more sophisticated version of the algorithm would not help in this example since it is impossible to allocate any of the flows in a lower indexed slot. After n iterations a flow in F_1 with grant size S_{max} arrives and must be rejected. The channel utilization at this point is

$$\begin{aligned}\eta = \frac{1}{I_K}\sum_{i=1}^{n} 2(S_{max} - i) + \frac{2n}{I_2} \quad &< \quad \frac{2n\,S_{max}}{I_K} + \frac{n}{I_1} \\ &= \quad \frac{2n\,S_{max}}{2^{K-1}\,n\,S_{max}} + \frac{n}{n\,S_{max}} = \frac{1}{2^{K-2}} + \frac{1}{S_{max}}\end{aligned} \tag{17}$$

Note that the channel utilization achieved by $PP - OLL$ in this example depends only on K and S_{max} and not on I_1. In this example the channel utilization achieved by OLL is at least $1 - KS_{max}/I_1 = 1 - K/n$. By choosing n, K and S_{max} to be large, we guarantee a high channel utilization for OLL while the channel utilization of $PP - OLL$, which does not depend on n, can be arbitrarily low.

Surprisingly, despite the poor worst case performance of $PP - OLL$, our simulation results (see subsection 9.4) indicate that on *average* its performance compares to that of OLL.

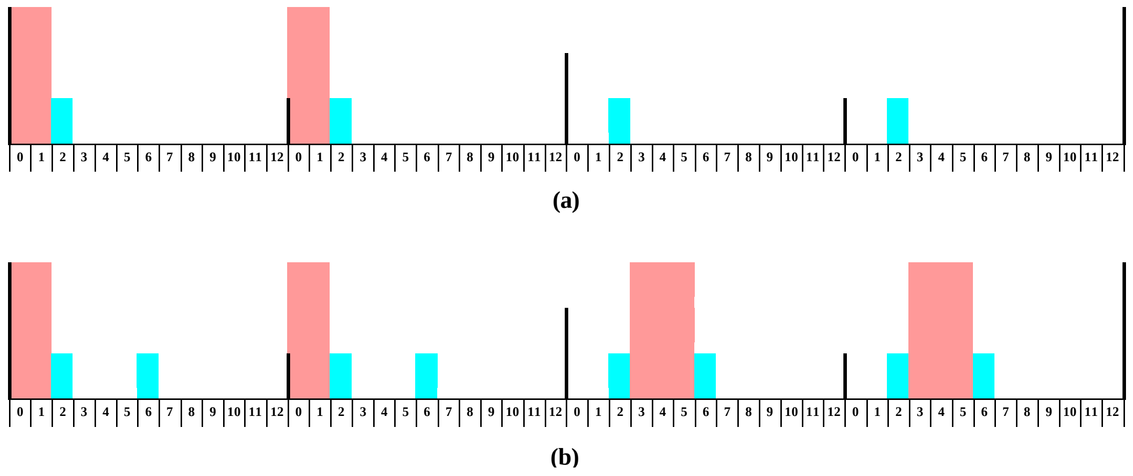

Figure 8: Example of an online schedule by algorithm $PP - OLL$. The Content of the first four bins is presented after the first iteration (a) and after the second iteration (b).

9.4 Expected Performance and Practical Implications

In the previous subsections we presented three online algorithms and provided lower bounds on their worst case performance. Theorems 7 and 8 provide a tight lower bound on the channel utilization achieved by algorithms $OPP - FF$ and OLL, respectively. However, the examples for which the lower bounds are tight require a particular set of sizes and arrival times which in practice are not likely to occur. In order to evaluate the expected performance of the algorithms we use simulation.

As an input to the algorithms we generate a random set of flows with grant intervals uniformly chosen from the set $\{I_1, \ldots, I_K\}$ and, independently, grant sizes uniformly distributed in $\{1, 2, \ldots, S_{max}\}$ (K and S_{max} are parameters of the simulation). In all the simulations we present here $I_1 = 100$ and $I_{j+1} = 2I_j$. Jitter constraints have been ignored in our simulations. We next apply the algorithms to the same randomly generated set of flows. To provide fair comparison with the worst case bounds, we stop the simulation when the first flow is rejected. It is therefore important to note that in practice the algorithms are expected to achieve a higher channel utilization since they may still accept flows after the first flow is rejected.

Our first observation is that simulation results can vary considerably between different runs of the simulation. This is illustrated in Figure 9 which shows the channel utilization achieved by algorithm OLL in 50 different simulation runs. The parameters for the simulation were $K = 10$ and $S_{max} = 0.5I_1$. The average channel utilization in this example is 0.5 and the standard deviation is 0.18.

Our second observation is that there is a considerable difference between the worst case bounds of Theorems 7 and 8 and the average channel utilization of the algorithms. This difference is especially noticeable in the performance of OLL. Figure 10 shows the average channel utilization of algorithms $OPP - FF$ and OLL along with their worst case bounds for $K = 5$ and $S_{max} \leq 0.5I_1$. Each point on the curves is the average of 500 simulation runs.

Figure 11 presents the average channel utilization achieved by algorithms $OPP - FF$, OLL, and $PP - OLL$ for values of $K = 5$ and $K = 10$. Recall that for $PP - OLL$ we showed that there are examples in which the channel utilization achieved by the algorithm is arbitrarily small. It is therefore interesting to observe that in our simulation there is no considerable difference in the performance of OLL and $PP - OLL$. When $S_{max} < 0.2I_1$ (which we consider typical) the

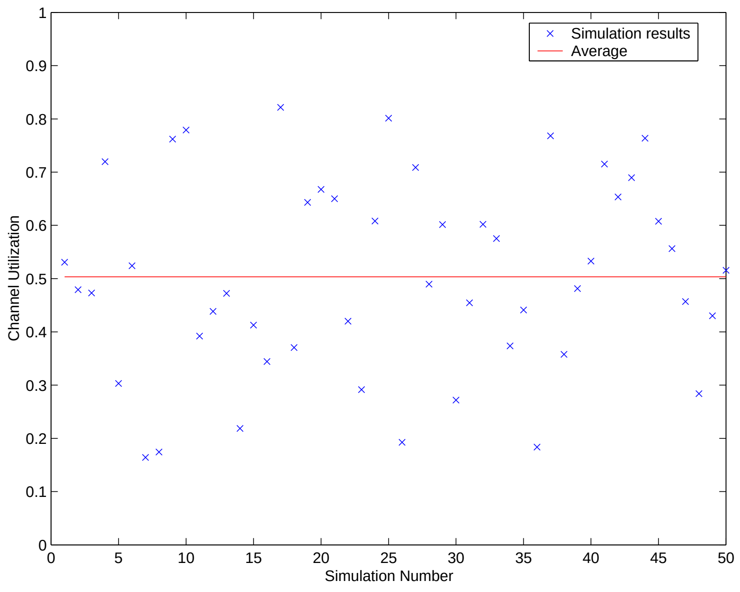

Figure 9: Channel utilization achieved by algorithm OLL in 50 different simulation runs when $K = 10$ and $S_{max} = 0.5I_1$.

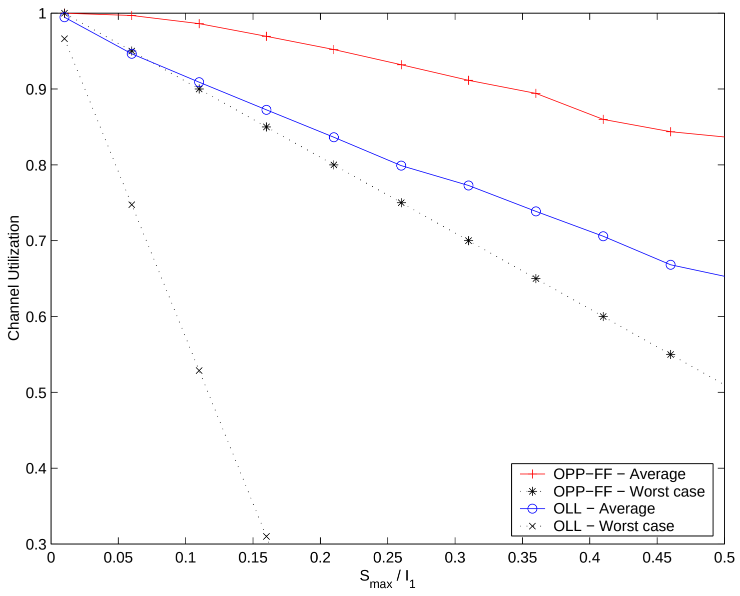

Figure 10: Average channel utilization achieved by algorithms $OPP - FF$ and OLL compared to their worst case bounds. Simulation results are for $K = 5$ and $S_{max} \leq 0.5I_1$.

difference in the channel utilization of the two algorithm is about 15% when $K = 5$ and about 30% when $K = 10$; when $S_{max} > 0.5I_1$ the two algorithms achieve almost the same channel utilization. Another interesting observation is that the performance of algorithm $OPP - FF$ is not effected by the value of K. This is in agreement with the bound of Theorem 7 which is also independent of K. As we expect, the performance of algorithm OLL decreases with K, a fact which is in agreement with the bound of Theorem 8. Figure 12 presents the average channel utilization of the three algorithms as a function of K when $S_{max} = 0.2I_1$. Note that the gap between the the performance of OLL and $PP - OLL$ increase as K increases.

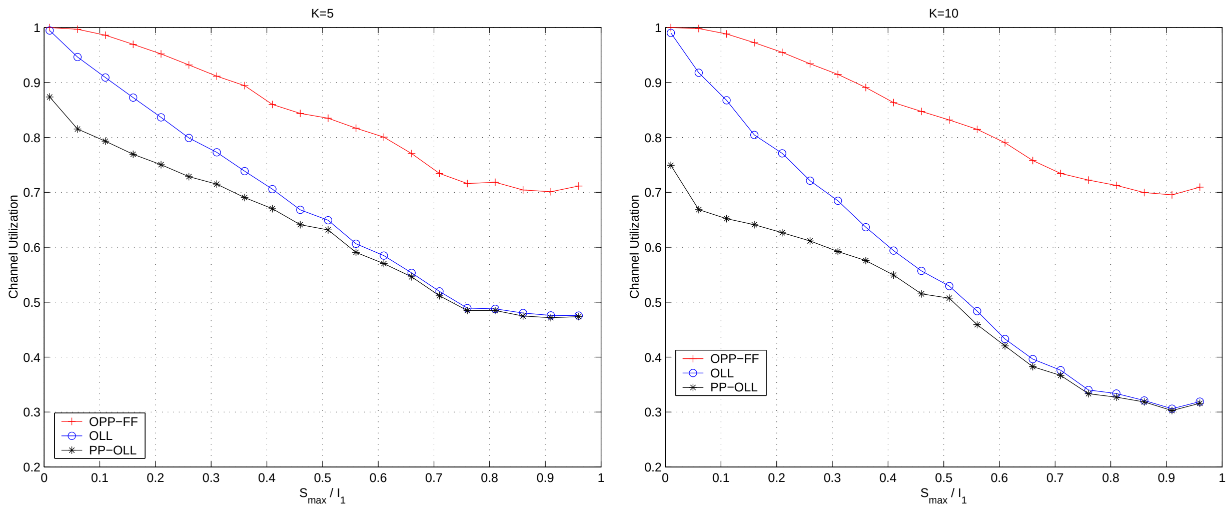

Figure 11: Average channel utilization achieved by algorithms $OPP - FF$, OLL, and $PP - OLL$ for values of $K = 5$ and $K = 10$.

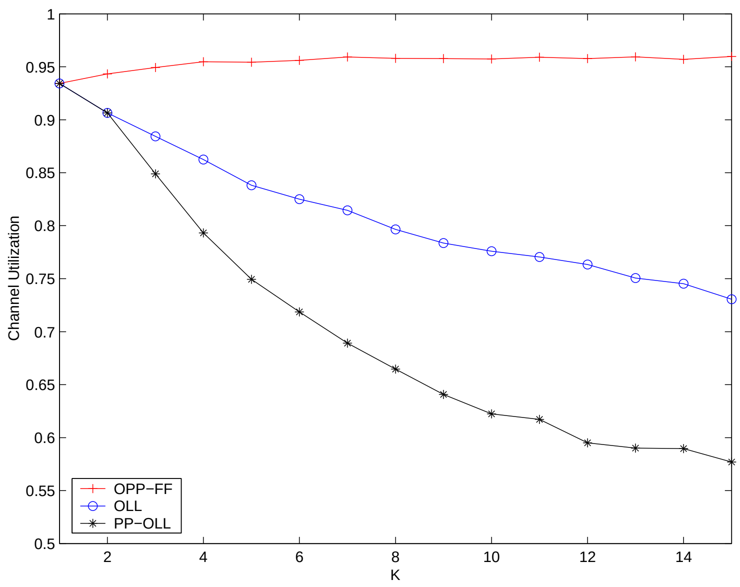

Figure 12: Average channel utilization of algorithms $OPP - FF$, OLL and $PP - OLL$ as a function of K when $S_{max} = 0.2I_1$.

9.4.1 Practical Implications

Let us now consider the practical implications of our results. For multimedia applications we may expect $K \leq 5$; a value of $K = 10$ is certainly sufficient since it provides a range of three order of magnitudes. Let us consider typical values of voice applications where $K = 5$, $I_1 = 10$ ms and $S_{max} = 200$ bytes. For a relatively slow link of 2 Mbps we have $S_{max} = 0.8$ ms, hence the lower bound of Theorem 8 implies that the channel utilization achieved by OLL is at least $\eta \geq 0.6$. The largest possible jitter is that of flows in F_2 and is no more than $(K-1)S_{max} = 3.2$ ms which is acceptable. From our simulation results we may conclude that in practice the average channel utilization OLL can achieve in this setting is around 0.9. If the tolerated jitter allows the use of $OPP - FF$ the channel utilization is guaranteed to be at least 0.92.

One way to improve the performance of online scheduling is to allow a tolerated jitter that enables the use of $OPP - FF$ (or any other version of an offline algorithm). Another possibility is to decrease the value of S_{max}. This can be done by changing the parameters of flows with large grant sizes, such that they are assigned smaller grant sizes at shorter intervals. The process is similar to the rounding procedure described in Section 8. In particular the process may incur a waste of bandwidth due the fixed size overhead in each packet. Another way to reduce S_{max} is to break large grant sizes into smaller grants such that the original flow is split into several sub flows (without violating jitter constraints). This method requires capabilities of splitting and reassembly and is expected to be beneficial only when the tolerated jitter is large.

9.5 Online Scheduling of Temporary Flows

The analysis we presented in the previous subsections considered permanent calls, i.e., calls with infinite duration. We now discuss the realistic case of temporary flows, that is, flows with a finite duration.

Defining a metric by which to evaluate the performance of an algorithm in the case of temporary flows is not as obvious as in the case of permanent flows. The reason is that the channel utilization is no longer monotonic increasing (non decreasing) in time. Simply comparing the channel utilization of a scheduling algorithm to that of an optimal algorithm at any point in time we choose does not provide meaningful results since the worst case performance ratio is likely to be unbounded. As an example, consider the time when one of the flow has been rejected by the scheduling algorithm but is accepted by an optimal algorithm; we now let all other flows end which results in an unbounded performance ratio. To overcome the problem we adopt a method, which is commonly used in analyzing scheduling problems (see e.g., [69], [66]), according to which the performance of an algorithm is taken to be the maximum channel utilization it achieves during the schedule.

Let us first consider the $OPP - FF$ algorithm. The algorithm is suitable for handling temporary flows. When a flow f_i ends, it is removed from the set F_c; when the next flow is established the new schedule created by $OPP - FF$ does not include f_i. We know that the algorithm does not reject a flow unless the channel utilization is more than $1 - (S_{max} - 1)/I_1$ and this property remains true for temporary flows. The only modification in the behavior of the algorithm is that it no longer maintains zero jitter for flows in F_1. The reason is that after a flow in F_1 ends, the new schedule created by $OPP - FF$ will shift the remaining flows in F_1 forward in time. To conclude, Theorem 7 remains valid for temporary flows with a modification requiring that $J_1 \geq I_1$ (as opposed to $J_1 = 0$ in the original theorem).

The situation is somewhat different for the OLL algorithm. Theorem 8 provides a lower bound on the channel utilization the algorithm is guaranteed to achieve before the first flow is rejected. This bound remains valid in the case of temporary flows. However, since OLL

does not reconstruct the whole schedule whenever a new flow arrives (as $OPP - FF$ does), the algorithm cannot guarantee that the same property holds when subsequent flows are rejected. In order to handle temporary flows OLL should group the grants of currently scheduled flows into a consecutive block of allocations. In order to reduce the jitter the algorithm may perform the operation of grouping flows only when a bin fills up. As a result of the need to group flows together the algorithm does no longer maintain the jitter properties of the permanent flows case. Theorem 8 should therefore be modified to require $J_j \geq I_1$ for $1 \leq j \leq K$ (recall that in the original theorem $J_j \geq (K-1)S_{max}$ was sufficient). By using a more sophisticated insert procedure we can probably reduce the jitter requirements by allocating new flows in the free spaces which are left when flows end.

Algorithm $PP - OLL$ has no performance guarantees even in the case of permanent flows. However, our simulation results indicate that in practice the algorithm is expected to perform reasonably. The algorithm as we presented it is not expected to maintain this performance in the case of temporary flows. The reason is that it only tries to insert a flow in the highest position which is available in all the bins the flow must be inserted into. In order to be able to reuse the slots that are freed when flows terminate, the algorithm must consider more positions from which to start scheduling a new flow. Note that even if the algorithm considers all I_1 positions the complexity of inserting a flow is only $O(m \cdot I_1)$ where m is the number of bins.

10 Scheduling CBR Flows over Multiple Channels

We consider the problem of scheduling CBR flows in networks with multiple channels. Our model of the problem is based on DOCSIS compliant cable networks where, in order to increase the overall bandwidth and improve noise immunity, each modem can access multiple upstream channels [1, 57]. In the general scheduling problem each flow is characterized by three parameters: grant interval I, grant size S, and grant jitter J. According to Theorem 1 the problem of scheduling flows with two or more different grant intervals is NP-hard even if the flows are to be scheduled on a single channel. We therefore restrict our attention to the case of uniform flows, i.e., when all the flows have the same values of I, S and J. The primary application we consider is packet telephony, hence we refer to such flows as *calls*.

In the case of uniform calls scheduling on a single channel is trivial. We divide the time axis of each upstream channel into frames of length I. Each call requires an allocation of a *time-slot* of length S in each frame. A call is accepted if there is a free time-slot in the frame (it can be assigned to any free time-slot) and is blocked otherwise. Note that in order to describe a schedule it is sufficient to describe the assignment of the calls in a single frame in each channel. What makes the scheduling problem interesting is the fact that each modem may establish several calls concurrently but all the calls of the same modem must be scheduled on a single upstream channel. This restriction is a result of the need to reduce hardware and scheduling complexities. The Headend can direct a modem to switch from one upstream channel to another, in which case all the ongoing calls of the modem must be scheduled on the new channel.

We now summarize the characteristics of the network and the variables we are going to use in our analysis.

- There are m identical channels.
- In each channel there are U time-slots per frame; establishing a call requires one time-slot per frame.
- Each modem may have $0 \leq c \leq c_{max}$ calls simultaneously. Unless otherwise stated we assume $c_{max} = U$.
- All the calls of the same modem must be scheduled on the same channel.
- It is possible to switch calls from one channel to another. When switching a call the tolerated jitter must not be violated.

For a given list of calls L we denote by $A(L)$ the number of calls that algorithm A accepts from L and by $OPT(L)$ the number of calls an optimal offline algorithm can accept. We define the performance ratio of algorithm A using equations (1)-(3).

We concentrate on the online scheduling problem, as this is the nature of the problem. We begin, however, with a short discussion on the offline problem.

10.1 Offline Problem

In the offline version of the problem the number of calls each modem has is known to the scheduling algorithm and all the calls are established at the same time. Note that in the offline problem channel switching is not required, hence jitter constraints are irrelevant and can be ignored. We define the decision version of the scheduling problem as follows: given a list of calls L, can all the calls in L be scheduled using m channels? In the case of a single channel the problem is trivial; a schedule exists if and only if the number of calls in L is no more than U. When we consider more than one channel the decision problem becomes NP-complete.

Theorem 9 *Deciding if a given set of uniform flows, can be scheduled on multiple channels is strongly NP-complete.*

Proof: We show a reduction from BIN PACKING (defined below) which is known to be strongly NP-complete [58].

BIN PACKING:

INSTANCE: A finite set A of integers (items) $a_1, a_2, ..., a_n$, a bin capacity $B \in Z^+$ and a positive integer k.

QUESTION: Can A be partitioned into k disjoint subsets (bins) $S_1, ..., S_k$ such that $\sum_{j \in S_i} a_j \leq B$ for $i = 1, ..., k$?

As the scheduling problem is almost identical to bin packing, the reduction is straightforward. We construct a scheduling problem with $m = k$ channels, $U = B$ time-slots per channel, and n modems. We denote by M_i the number of calls of modem i and choose an instance in which $M_i = a_i$, $1 \leq i \leq n$. For any "yes" instance of BIN PACKING we create a legal schedule as follows. If item a_i is packed in S_j then all the calls of modem i are scheduled over channel j. Recall that in a legal schedule all the calls of a modem must be scheduled over the same channel. Hence, for any "yes" instance of the scheduling problem we can create a valid bin packing by placing item a_i in S_j iff the calls of modem i are scheduled over channel j. Thus, the answer to the scheduling problem is "yes" iff the answer to BIN PACKING is "yes". ■

We model the scheduling problem as a variant of bin packing. The channels correspond to the bins; there are m bins each of size U. The modems correspond to the items; the size of an item is the number of calls the modem has. Since we consider the offline problem the tolerated jitter has no effect. The goal is to schedule (pack) as many calls as possible. The scheduling problem is different from classical bin packing since it allows to pack part of an item (which means that only part of the calls the modem requested were accepted). As an example consider the FF algorithm with $m = 2$, $U = 10$ and a list of items $L = \{1, 1, 9, 9\}$. In bin packing two items of size 1 are packed in the first bin and one item of size 9 in the second bin; the last item is discarded. The performance ratio is $R_{FF}(L) = \frac{20}{11}$. In the scheduling problem the last item may be partially packed in the first bin, i.e., 8 out of 9 calls are assigned to the first channel. The performance ratio is therefore much better $R_{FF}(L) = \frac{20}{19}$.

We now consider the Next-Fit (NF) algorithm for the offline scheduling problem and estimate its performance.

NF **Algorithm**: We present two variants of NF for the problem. In both variants the algorithm keeps only one open bin and once a bin is closed it is not used again.

1. The algorithm keeps one open bin and packs items into the open bin until the next item does not fit; the open bin is then closed and the item is packed in a new bin. After all bins have been closed the algorithm stops. As a worst case example we choose a list of $2m$ items arranged in the following way $L = \{\frac{U}{2}, 1, ..., \frac{U}{2}, 1\}$. We have $NF = m(\frac{U}{2} + 1)$, $OPT = mU$, $R_{NF}(L) = \frac{2U}{U+2}$.

2. The algorithm packs items into the open bin until the bin is full. The last item to be packed in each bin may be partially packed. The algorithm then moves to the next bin. As a worst case example we choose $m = 2U - 1$ and the following list $L = \{1, 1, ..., 1, U\} \times U$, i.e., the list is made of U sets of $U - 1$ items of size 1 followed by one item of size U. In this example $A = U^2$, $OPT = (2U - 1)U$, $R_{NF}(L) = \frac{2U-1}{U}$.

Standard results of bin packing tell us that the performance ratio of NF is not more than 2 [62]. The above examples therefore prove that the asymptotic worst case performance ratio of NF in both cases is $R_{NF} = 2$.

10.2 Online Problem

We now consider the online version of the scheduling problem where calls are established over time. We distinguish between the case of permanent calls in which calls do not terminate (i.e., have infinite duration) and the case of temporary calls where calls may terminate over time. Since the offline problem is NP-hard we concentrate on approximation algorithms. A scheduling algorithm must provide an answer to the following questions:

- When the first call of a modem is established
  - In what channel to allocate the call?
  - In which time-slot in the selected channel to allocate the call?
- When a new call is added to an active modem, in what time-slot should the call be allocated?
- In case a modem is to be switched to a different channel
  - To which channel should the modem be switched?
  - In what time-slots in the selected channel to allocate the calls of the modems such that the jitter constraint of all the ongoing calls is not violated?

In this work we consider three algorithms. The algorithms specify the channel selection rules; in all three algorithm a new call is scheduled in the first available time-slot in the selected channel.

1. First Fit (FF) - Channels are numbered $1, ..., m$. The first call of a modem is scheduled on the first available channel. Subsequent calls are scheduled on the channel the modem is currently occupying. If the channel is full all the calls of the modem are switched to the first channel that can accommodate them. If there is no such channel the call is blocked.
2. Best Fit (BF) or Most Loaded - The first call of a modem is scheduled on the most loaded channel, i.e., the channel with the largest number of calls, (ties broken by lower index). Subsequent calls are scheduled on the channel the modem is currently occupying. If the channel is full all the calls of the modem are switched to the most loaded channel that can accommodate them. If there is no such channel the call is blocked.
3. Worst Fit (WF) or Least Loaded - The first call of a modem is scheduled on the least loaded channel (ties broken by lower index). Subsequent calls are scheduled on the channel the modem is currently occupying. If the channel is full all the calls of the modem are switched to the least loaded channel that can accommodate them. Otherwise the call is blocked.

10.2.1 Permanent Calls with no Jitter Constraint

In this subsection we assume $J = I$ which means that there is no jitter constraint when switching calls from one channel to another. We first provide an upper bound on the performance ratio of any deterministic online algorithm.

Lemma 2 *For the scheduling problem with permanent calls and no jitter constraint the performance ratio of any deterministic online algorithm A satisfies $R_A \geq \frac{3}{2} - \frac{3}{4U+2}$.*

Proof: Consider first a family of algorithms that do not reject a call if it can be accepted. We choose $U \gg 1$ to be an even number and $m = 3$ channels. There are four modems $M_1 - M_4$ and each modem initially has $U/2$ calls. Regardless of the way the calls have been scheduled we know that one of the channels is full with the calls of two different modems; without loss of generality, let us assume it is channel 1 and the modems are M_1 and M_2. We now let one more call arrive to M_3. At this point we have the following configuration: M_1 and M_2 occupy channel 1 while M_3 and M_4 occupy two different channels (see Fig. 13.a). We now let M_1 and M_2 receive $U/2$ more calls each. These calls are all blocked by the algorithm, hence $A(L) = 2U + 1$. An optimal algorithm can accept $OPT(L) = 3U$ calls (see Fig. 13.b). The ratio in this example is $R_A(L) = \frac{3}{2} - \frac{3}{4U+2}$.

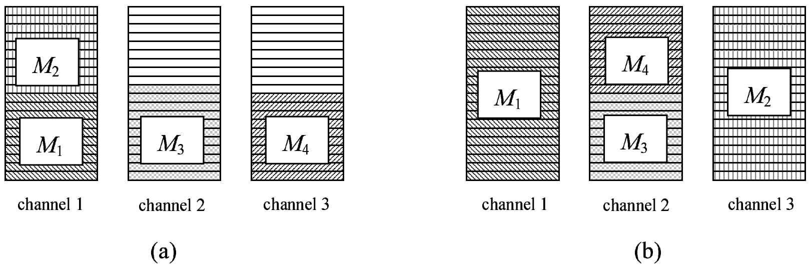

Figure 13: Schedules of algorithm A (a) and an optimal algorithm (b) for the above example.

To complete the proof consider the case where the algorithm may reject a call that could be accepted. Suppose a call arrives to modem X and is rejected although it could be accepted. We repeat sending calls to modem X until it accepts a call, in which case we continue as in the above example, or until U calls have arrived to modem X. The calls that have been rejected could be accepted by an optimal algorithm so the performance ratio can only degrade. ■

Theorem 10 *For algorithm $A \in$ {Worst-Fit, Best-Fit, First-Fit} with no jitter constraint the worst case performance ratio is $R_A = \frac{3}{2}$.*

Proof: Lemma 2 provides a lower bound on the performance ratio of the algorithms (when $U \to \infty$). We now show that this is also the upper bound. To do so we evaluate the difference between the number of calls algorithm A accepts and the number of calls an optimal algorithm can accept. We assume A has scheduled several calls and consider the moment in which the first call is blocked by A. Assume this call belongs to modem X_1 and the modem has x_1 ongoing calls at that time. We recognize the following properties:

1. The channel containing the calls of modem X_1 is full (otherwise the new call is accepted by A). This channel must also contain at least one call belonging to a different modem (since $x_1 < U$).
2. The content of any other channel is at least $U - x_1$ (otherwise modem X_1 is switched to a different channel and the new call is accepted).
3. An optimal algorithm can accept at most $U - x_1$ more calls to modem X_1.

We know that the channel modem X_1 is occupying has calls of at least one more modem. Note that by Property 2, if $x_1 \le \frac{U}{3}$ the content of all the channels is at least $\frac{2U}{3}$, hence the performance ratio is at most $\frac{3}{2}$. We conclude that, in a worst case example, in addition to modem X_1 the channel has one more modem that would have blocked calls. Assume the next modem to receive a call which is blocked is modem X_2 and it has x_2 accepted calls. Clearly the same properties we listed above must also hold for X_2. Let $x = \min\{x_1, x_2\}$. The number of calls accepted by algorithm A and an optimal algorithm satisfies

$$\begin{aligned} A(L) &\geq U + (m-1)(U-x) \\ OPT(L) &\leq \min\{mU,\ U + (m-1)(U-x) + 2(U-x)\} \end{aligned} \tag{18}$$

The performance ratio satisfies the following:

$$R_A \leq \frac{\min\{mU,\ U + (m+1)(U-x)\}}{U + (m-1)(U-x)} \tag{19}$$

The maximum possible value of the expression in (19) is $R_A = \frac{3}{2}$; it is obtained when $x = \frac{U}{2}$ and $m = 3$. This provides the necessary upper bound on the performance ratio of the algorithms. ■

Theorem 10 provides the worst case performance ratio of the algorithms we consider for certain values of U and m. We can extend the example in Lemma 2 for any $m = 3k$ (k an integer). In other cases the performance ratio is slightly lower and depends on U.

10.2.2 Permanent Calls with no Jitter Constraint and no Channel Switching

To evaluate the benefits of channel switching we now analyze the scheduling problem under the assumption that channel switching is not allowed, i.e., a modem is assigned a channel when the first call is established and all subsequent calls must be assigned to that channel.

A worst case example for algorithms FF and BF is straightforward. The first U calls are for different modems and they are all assigned to the first channel. In the next stage each modem receive $U-1$ additional calls. Since the channel is full and channel switching is not allowed all the calls are blocked. We therefore have $FF(L) = BF(L) = U$ and $OPT = \min\{U^2, mU\}$; the performance ratio is $R_{FF} = R_{BF} \geq \min\{m, U\}$. As we can see the performance ratio increases with both U and m. When $m \leq U$ the performance ratio is the worst possible (since at least one channel is full before calls are blocked).

The WF algorithm performs better in the case where channel switching is not allowed. However, we now show that the performance ratio is monotonically increasing with U and therefore tends to m as U increases. Consider the following example: in the first stage $k \cdot m$ calls for different modems arrive; as a result each channel is assigned k calls. In the next stage $U-k$ calls of a single modem arrive and are allocated in the first channel; at this point the first channel is full. We now have $U-1$ calls arriving to each modem in the first channel; since the channel is full all these calls are blocked. The algorithm accepts $WF(L) = U + (m-1)k$ calls while an optimal algorithm can accept $OPT(L) = \min\{(k+1)U + (m-1)k,\ mU\}$. For simplicity we can choose $k = m$ and $U > m^2$ in which case the ratio is

$$R_{WF} = \frac{mU}{U + (m-1)m} = m - \frac{(m-1)m^2}{U + (m-1)m} \tag{20}$$

As U increases the performance ratio converges to m.

From the above examples we conclude that channel switching considerably improves the worst case performance ratio of the scheduling algorithms. When channel switching is disabled the performance ratio is monotonically increasing with U and m; when channel switching is enabled the performance ratio is constant.

10.2.3 Temporary Calls with no Jitter Constraint

In this section we consider the case where calls begin and end over time. In addition to I, S, J and the call's starting time, now each call also has a duration. The information about the duration of a call is unknown to the algorithm when the call is established. We assume that a call that has been accepted may not be interrupted.

Defining a way in which to evaluate the worst case performance of an algorithm for temporary calls is problematic. Suppose we use our current definition and evaluate the performance by the number of calls the algorithm accepts. In this case it is always possible to come up with an example that leads to an unbounded performance ratio. Clearly calls which the algorithm does not accept but are accepted by the optimal algorithm may end and new calls with the same parameters may immediately begin; repeating this pattern result in increasing the performance ratio hence the performance ratio tends to infinity. Note that the same applies if we evaluate the performance of an algorithm by its call blocking probability, as is typically done in average case analysis [57].

To overcome the problem we define, as we did in Section 9, the performance of an algorithm as the maximum number of calls it schedules at some time during its execution. Using this definition the worst case results for permanent calls are also valid for temporary calls. The reason is that our bounds on the worst case performance ratio were all achieved by evaluating the number of calls the algorithm schedules when the first call is blocked.

10.2.4 Permanent Calls with Jitter Windows

In this section we evaluate the use of jitter windows as a means for maintaining jitter constraint when switching a modem from one channel to another. The use of jitter windows has been proposed in [57]. According to this proposal a frame containing U time-slots is divided into W windows such that the length of a window is less than the tolerated jitter of the calls, i.e., $\frac{I}{W} \leq J$. Hence, a call can be moved freely within its jitter window without violating jitter constraints. Jitter windows provide a simple way to ensure that when a modem is switched form one channel to another the jitter constraint of every call is not violated. It is important to note, however, that jitter windows add restrictions to the scheduling algorithm which are not imposed by the scheduling rules.

For simplicity we assume that U is a multiple of W; hence each window can hold up to U/W calls. A modem may use only one channel and within this channel it may have calls in different jitter windows. When a modem is switched to a different channel each and every call in the original channel must be allocated in the same jitter window in the new channel.

We first establish an upper bound on the worst case performance ratio of any algorithm that allows channel switching.

Lemma 3 *Let A be an algorithm that allows channel switching and uses jitter windows. If A does not block calls unnecessarily the worst case performance ratio of A satisfies*

$$R_A \leq W \quad \textit{for every } W \geq 2. \tag{21}$$

Proof: We assume nothing about the way A schedules the calls. Our only assumption is that when a modem tries to establish a call the call is accepted in two cases: 1) The current channel the modem is occupying has a free time-slot, or 2) The call can be accepted if the modem is switched to another channel.

Consider a call that arrives to modem X and is blocked by A. Suppose modem X has x ongoing calls in jitter window W_j at that time. We recognize the following properties:

1. The channel containing the calls of modem X is full.
2. The content of W_j in all the other channels is at least $\frac{U}{W} - x$.
3. An optimal algorithm can accept at most $U - x$ more calls to modem X.

Algorithm A may block calls of other modems in the channel modem X is occupying. We denote by k the number of modems with blocked calls. Note that if a jitter window has two modems with blocked calls the content of the same jitter window in any other channel is at least $\frac{U}{2W}$. It follows that if there is more than one jitter window with two or more modems with blocked calls in it, the performance ratio is less than W. We may therefore assume that one jitter window, say W_1, has $1 \leq k_1 \leq \frac{U}{W}$ modems with blocked calls and in $0 \leq k_2 \leq W - 1$ other windows there is one modem with blocked calls. As a result, the number of calls accepted by A and OPT satisfy

$$\begin{aligned} A(L) &\geq U + (m-1)\left(\tfrac{(k_1-1)U}{k_1 W} + k_2\right) \\ OPT(L) &\leq min\{mU,\ A + (k_1 + k_2 - 1)U\} \end{aligned} \tag{22}$$

Simple analysis of the ratio $R_A = \frac{OPT(L)}{A(L)}$ shows that the worst case performance ratio satisfies $R_A < W$ for any set of values of U, m, k_1, and k_2. It converges to W in two cases

1. $k_1 = 1,\ k_2 = W - 1,\ U \to \infty$.
2. $k_1 = \frac{U}{W},\ k_2 = 0,\ U \to \infty$.

The above result holds for any algorithm regardless of the way the calls are scheduled. ■

Let us now consider the Worst-Fit, Best-Fit and First-Fit algorithms. We first define the algorithms for the case of jitter windows.

Worst-Fit (Best-Fit) with Jitter Windows: The algorithm uses the least (most) loaded rule. It assigns the first call of a modem to the least (most) loaded channel and within that channel the call is assigned to the least (most) loaded jitter window. If a modem already has an ongoing call a new call is assigned to the least (most) loaded jitter window in the channel the modem is occupying. If that channel is full the algorithm tries to switch all the calls of the modem to a different channel. If there are several channels that can accommodate the calls of the modem, the least (most) loaded channel is selected.

First-Fit with Jitter Windows: The algorithm assigns the first call of a modem to the first available channel and within that channel the call is assign to the first available jitter window. If a modem already has an ongoing call a new call is assigned to the first available jitter window in the channel the modem is occupying. If that channel is full the algorithm tries to switch all the calls of the modem to a different channel. It examines the channel according to their order and select the first that can accommodate the calls of the modem.

Theorem 11 *For algorithm $A \in \{$Worst-Fit, Best-Fit, First-Fit$\}$ using jitter windows the worst case performance ratio is $R_A = W$ for every $W \geq 2$.*

Proof: Lemma 3 provides an upper bound on the performance ratio of the algorithms. We present an example for each algorithm that proves the lower bound.

Worst case example for Worst-Fit
We create a situation where the first channel is full such that each jitter window is fully occupied (has $\frac{U}{W}$ calls) by a different modem. In the rest of the channels there is one call belonging to

a different modem in each window (see Figure 14). This situation is created if the first mW calls belong to different modems and afterwards calls to the modems occupying the first channel arrive one after the other until the channel is full. At this point we let each modem in the first channel receive $U - \frac{U}{W}$ more calls. All these calls are blocked by the algorithm since there is no way to move the existing calls to the same jitter window in any other channel. The number of calls that WF accepts is $WF(L) = U + (m-1)W$ while an optimal algorithm can accept $OPT(L) = min\{mU, WU + (m-1)W\}$. We choose $U > W(W+1)$ and $m = W+1$ which means that all the calls are accepted by an optimal algorithm. The performance ratio in this example is

$$R_{WF} = \frac{WU + (m-1)W}{U + (m-1)W} = W\left(\frac{U+W}{U+W^2}\right) \tag{23}$$

As $U \to \infty$ the worst case performance ratio converges to $R_{WF} = W$.

Using Lemma 3 and the above example we can derive bounds on the performance ratio for any set of values of U, m and W.

$$\frac{min\{WU + (m-1)W,\ mU\}}{U + (m-1)W} \leq R_{WF} \leq W \tag{24}$$

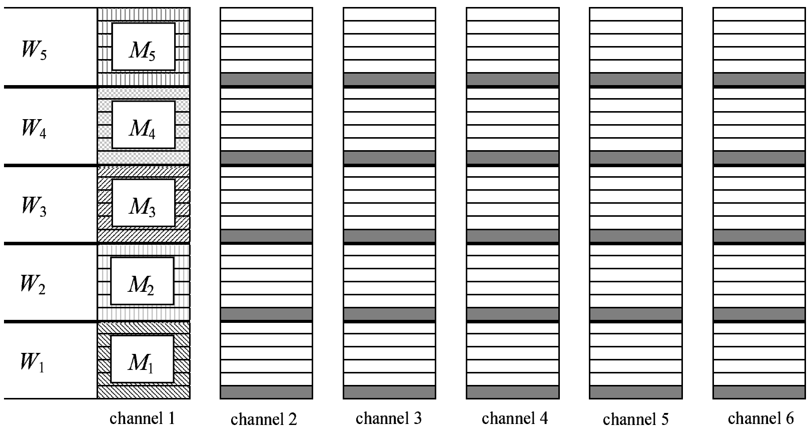

Figure 14: Worst case example of algorithm WF for $U = 30$ and $W = 5$. Each modem $M_1, ..., M_5$ now receive 14 more calls.

Worst case example for Best-Fit and First-Fit

We create a situation where the first channel is full and its first jitter window contains $\frac{U}{W}$ calls belonging to different modems. In the rest of the channels the first jitter window is full containing $\frac{U}{W}$ calls of the same modem (see Figure 15). At this point we let each modem in the first jitter window of the first channel receive $U-1$ more calls. All these calls are blocked by the algorithm since there is no way to move the existing calls to the same jitter window in any other channel. The number of calls that BF or FF accept is $BF(L) = FF(L) = U + (m-1)\frac{U}{W}$ while an optimal algorithm can accept $(U-1)\frac{U}{W}$ additional calls, i.e., $OPT(L) = U + (m-1)\frac{U}{W} + (U-1)\frac{U}{W}$. We let the number of channels be $m = \frac{U}{W-1} + 1$ (we choose U for which m is an integer). The performance ratio in this case is

$$\begin{aligned} R_{BF} &= R_{FF} = \frac{U + (m-1)\frac{U}{W} + (U-1)\frac{U}{W}}{U + (m-1)\frac{U}{W}} \\ &= \frac{W + \frac{U}{W-1} + U - 1}{W + \frac{U}{W-1}} > W\left(\frac{U}{U+W}\right) \end{aligned} \tag{25}$$

As $U \to \infty$ the worst case performance ratio converges to W.

Using Lemma 3 and the above example we can derive bounds on the performance ratio for any set of values of U, m and W.

$$\frac{\min\{U+W+m-2,\ mW\}}{W+m-1} \leq R_{BF} = R_{FF} \leq W \tag{26}$$

We now show how the above example is created. Denote by $C = \frac{U}{W}$ the number of time-slots in a jitter window and let $M = \{M_1, M_2, ..., M_C\}$ be a set of C different modems. We start with channel 1 containing the following set of calls: in the first window there is one call for each modem in M; the rest of the channel is full with the calls of modem M_0. Next modem M_C receives a call so it is switched to channel 2. Now $C-2$ calls arrive to M_C and fill the first window of channel 2. Next a call arrives to modem M_{C+1} and is assigned to the first jitter window of channel 1; the next call is also to M_{C+1} and as a result the modem is switched to the first jitter window of channel 3. We now let modem M_{C+1} receive $C-2$ calls so the first jitter window of channel 3 is also full. The process continues until the first jitter window of all the channels is full. At this stage we arrived to the situation assumed by the worst case example we presented. ■

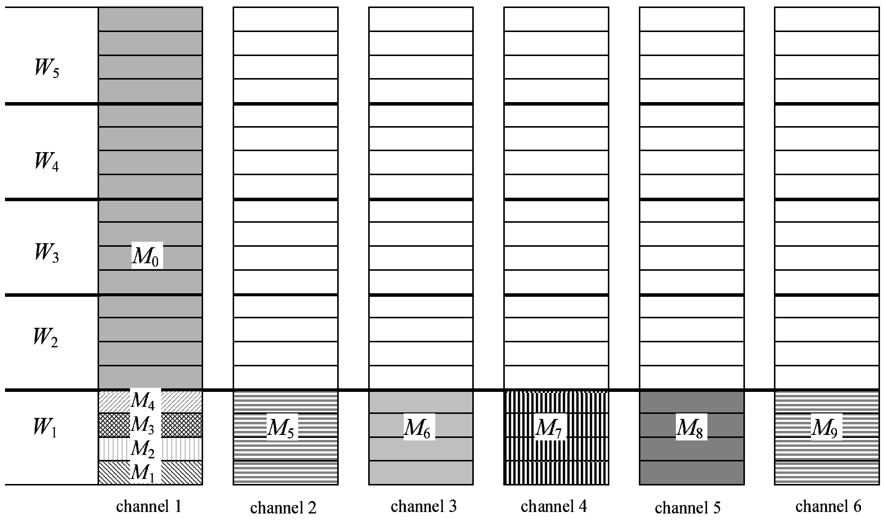

Figure 15: Worst case example of algorithms BF and FF for $U = 20$ and $W = 5$. Each modem $M_1, ..., M_4$ now receive 19 more calls.

10.3 Jitter Constraint without Using Jitter Windows

The use of jitter windows simplifies the scheduling algorithms. However, jitter windows add restrictions which are not imposed by the scheduling rules and may therefore degrade the performance of an algorithm. In this section we analyze the performance of our algorithms when jitter windows are not used. To provide a comparison with the case where jitter windows are used we define $W = \frac{U}{J}$. We show that as U increases the worst case performance ratio of the algorithms converges to W.

The example we present is suitable for all three algorithms. We create a situation in which the first channel is full containing k calls of different modems $(M_1, ..., M_k)$ and $U-k$ calls of modem M_{k+1}. In each of the other channels there are $\frac{U}{W} + k$ calls of a single modem. It is easy to verify that this initial situation is possible in all three algorithms. In the next stage

$U-k$ calls arrive to each modem in the first channel, i.e., $M_1, ..., M_{k+1}$. Since there is no way to switch a modem from channel 1 without violating its jitter constraint all these calls are blocked. The algorithm accepts $A = U + (m-1)(\frac{U}{W} + k)$ calls while an optimal algorithm can accept $OPT = \min\{(k+1)U + (m-1)(\frac{U}{W} + k),\ mU\}$. If we choose $k = \sqrt{U}$ (assuming it is an integer) and $m = \sqrt{U} + 1$ the performance ratio is

$$R_A = \frac{(\sqrt{U}+1)U}{U + \sqrt{U}\left(\frac{U}{W} + \sqrt{U}\right)} = \frac{W(\sqrt{U}+1)}{\sqrt{U} + 2W} \tag{27}$$

As U increases the performance ratio converges to W.

10.4 Bounded Number of Calls per Modem

In this section we assume the number of simultaneous calls a modem may have is bounded. The motivation for exploring the problem is that in practice each modem can only have a limited number of simultaneous calls. We denote by c_{max} the maximum number of simultaneous calls each modem may have.

We consider here the case of no jitter constraint and permanent calls. We construct a worst case example in which the first channel is filled with calls of $\frac{U}{x}$ different modems, each with x calls ($x \leq c_{max}$ is an optimization parameter). The content of the rest of the channels is $U - x$. we now let each modem in the first channel receive $c_{max} - x$ calls which are blocked by the algorithm. The number of calls the algorithm accepts is $A = U + (m-1)(U-x)$ while an optimal algorithm can accept $OPT = U + (m-1)(U-x) + (c_{max} - x)\frac{U}{x}$. We choose the number of channels to be $m = \frac{U(c_{max}-x)}{x^2} + 1$ so $OPT = mU$. The performance ratio in this case is

$$R_A = \frac{mU}{mU - (m-1)x} = \frac{U(c_{max} - x) + x^2}{2x^2 + U(c_{max} - x) - c_{max}x} \tag{28}$$

The maximum is found for

$$x = \frac{c_{max}\sqrt{U}}{\sqrt{U} + \sqrt{c_{max}}} \tag{29}$$

Since x must be an integer we can establish the following lower bound on the performance ratio

$$R_A \leq 1 + \frac{c_{max}}{U - c_{max} + 2\sqrt{U\, c_{mac}}} \tag{30}$$

As we expect when U increases the performance ratio converges to 1 since each channel must contain at least $U - c_{max}$ calls before the first call is blocked. When $c_{max} = U$ we get the result we obtained in subsection 10.2.1.

11 Conclusions and Further Research

This work addressed the problem of scheduling CBR flows over a slotted TDMA channel. A CBR flow is characterized by its grant size, grant interval, and grant jitter. We focused on the case where grant intervals are an integer multiple of each other. In this setting the scheduling problem can be modelled as a variant of bin packing where bin sizes can be modified in a constraint manner. The general scheduling problem was proved to be NP-hard even when there are only two different grant intervals. We developed several scheduling algorithms and investigated their performance. In the offline problem when there are K related grant intervals and the tolerated jitter of the flows satisfies $J_{min} \geq (K-1)(S_{max}-1)$ the $FFJ-K$ algorithm provides an optimal solution. If the schedule should be perfectly-periodic one can use the $PP-FF$ algorithm which guarantees that the channel utilization is at least $1-S_{max}/I_1$. For the online problem we suggested three algorithm; each algorithm is suitable for a different category of jitter requirements. Larger tolerated jitter allows us to design more efficient online algorithms. When the tolerated jitter of each flow is at least its grant size we can use an offline algorithm and match the performance of algorithm $PP-FF$. When the tolerated jitter is more restricted we can use algorithm OLL which guarantees that the channel utilization is at least $1-KS_{max}/I_1$.

We expect our algorithms to perform well when applied to scheduling flows of interactive multimedia applications such as VoIP. The reason is that these applications use relatively small packet sizes (hence $S_{max} \ll I_1$), and can tolerate jitter which is on the order of the grant interval.

There are several subjects that await further research. We provided analytic average case analysis only for the NFJ algorithm. It is interesting to analyze the expected performance of other algorithms we presented, in particular the online algorithms. This work is mainly concerned with the case of a single channel. For multiple channels our analysis covers the case of identical flows only. More results are needed for the problem of scheduling flows with different parameters over multiple channels. Another interesting problem is that of scheduling real-time Variable Bit Rate (VBR) flows. VBR flows have fixed grant interval but variable grant sizes. One way of scheduling VBR flows is to treat them as CBR flows where the grant size of the CBR flow is taken to be the maximum grant size of the VBR flow. However, this method may result in poor channel utilization since there can be a considerable difference between the maximum and the average grant size. VBR applications (such as compressed video) can typically tolerate a relatively large jitter which gives hope that efficient algorithms can be developed for the problem.

APPENDIX

A Effects of Jitter on the Scheduling Problem

In the previous sections we analyzed the scheduling problem by converting it into a variant of bin packing in which bin sizes can be modified. In order to change a bin's size we displace the grants to the flows that determined the bin's limit. The amount by which we can displace these grants depends on the tolerated jitter of the flows. Let us define the *effective jitter* to be the maximum allowed displacement of a group of flows, from their nominal grant times. In this section we elaborate more on the effective jitter and how it effects the performance of scheduling algorithms.

A.1 Determining the Effective Jitter

Our goal is to determine the effective jitter for a group of flows which are allocated in one consecutive block. The flows may different parameters of grant interval, grant size and grant jitter. We know it is possible to move each flow forward in time up to its grant jitter. The effective jitter for a group of flows in a single block, is the amount by which we can move the entire block forward in time. Let us consider the case of two intervals. The flows in F_1 are scheduled in blocks, i.e., over consecutive slots, creating fixed size bins. We want to know in what way we can modify the bin sizes. There are two immediate bounds; the lower bound is the minimal grant jitter among the flows in F_1 while the upper bound is the maximal grant jitter. If all flows have the same jitter J then J is clearly the effective jitter. When flows have different values of grant jitter, finding the effective jitter is more difficult. Consider the following example: we have two flows in F_1 with the same grant size, $S(1) = S(2) = 1$, but different jitter $J(1) = 4$, $J(2) = 1$. The grant interval is $I = 5$ so initially we have bins of size 3. We can clearly create bins of size 4 by moving the two flows one slot forward in time. Can we create bins of sizes larger than 4? It turns out that in this example the answer is positive, we show below how we can get bins of sizes 5 and 1, so the effective jitter is 2.

1	2						2	1		1	2				1	2	

Note that in order to get an effective jitter which is larger than the minimum jitter in F_1 we had to reorder the flows. In many cases it is not possible to increase the effective jitter. For example if we take the previous example with $S(1) = 2$ the effective jitter is 1. As we can see, determining the effective jitter is not always easy. For this reason we assumed throughout the paper that the effective jitter equals the minimal grant jitter among all the flows in the group. Although this assumption may be conservative in some cases, it simplifies the algorithms (and the analysis) considerably. Using this assumption ensures that every block can be moved forward in time by the effective jitter.

References

[1] Cable Television Laboratories Inc. CableLabs, *Data-Over-Cable Service Interface Specifications - Radio Frequency Interface Specification (status: interim)*, April 2000.

[2] "Cable television laboratories inc. CableLabs." `http://www.cablelabs.com`.

[3] J. Albrycht, "VoIP: putting the pieces together," *CED*, vol. 24, pp. 44–48, December 1998.

[4] B. Beser, "Providing enhanced services over DOCSIS," *CED*, vol. 27, pp. 80–84, April 2001.

[5] Cable Television Laboratories Inc. CableLabs, *PacketCable^{TM} Dynamic Quality-of-Service Specification (status: interim)*, August 2000.

[6] T. H. Cormen, C. H. Leiserson, and R. L. Rivest, *Introduction to Algorithms.* Cambridge, MA, USA: MIT Press, 1990.

[7] J. Stankovic, M. Spuri, M. Di-Natale, and G. C. Buttazzo, "Implications of classical scheduling results for real-time systems," *IEEE Computer*, vol. 28, pp. 16–25, June 1995.

[8] E. L. Lawler, "Recent results in the theory of machine scheduling," in *Mathematical Programming: the State of the Art* (A. Bachem, M. Groetschel, and Korte, eds.), pp. 202–233, Berlin: Springer, 1983.

[9] J. Stankovic and K. Ramamritham, *Advances in Hard Real-Time Systems.* IEEE Computer Society Press, 1993.

[10] H. Chetto, M. Silly, and T. Bouchentouf, "Dynamic scheduling of real-time tasks under precedence constraints," *Real Time Systems*, vol. 2, no. 3, pp. 181–194, 1990.

[11] R. S. Oliveira and S. Fraga, "Fixed priority scheduling of tasks with arbitrary precedence constraints in distributed hard real-time systems," *Journal of Systems Architecture*, vol. 46, no. 11, pp. 991–1004, 2000.

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

[13] A. K. Mok, *Fundamental Design Problems of Distributed systems for the Hard Real-Time Environment.* Phd dissertation, Massachusetts Institute of Technology, Cambridge, MA, 1983.

[14] J. Lehoczky, L. Sha, and Y. J. Ding, "The rate monotonic scheduling algorithm: exact characterization and average case behavior," in *Proceedings of Real Time Systems Symposium*, (Los Alamitos CA USA), pp. 166–171, IEEE, 1989.

[15] J. Y. T. Leung and J. Whitehead, "On the complexity of fixed-priority scheduling of periodic, real-time tasks," *Performance Evaluation*, vol. 2, pp. 237–250, December 1982.

[16] K. Jeffay, D. F. Stanat, and C. U. Martel, "On non-preemptive scheduling of period and sporadic tasks," in *Proceedings 12st Real-Time Systems Symposium*, (Los Alamitos, CA, USA), pp. 129–139, IEEE, 1991.

[17] A. Khil, S. Maeng, and J. Cho, "Non-preemptive scheduling of real-time periodic tasks with specified release times," *IEICE Transactions on Information and Systems*, vol. 5, pp. 562–572, 1997.

[18] A. Guntsch, "Adaptive edf non-preemptive scheduling for periodic tasks in the hard real-time systems," in *Proceedings of the ISCA 15th International Conference Computers and Their Applications*, (Cary, NC, USA), pp. 361–367, 2000.

[19] J. Stankovic, M. Spuri, K. Ramamritham, and G. C. Buttazzo, *Deadline Scheduling for Real-time systems: EDF and Related Algorithm.* Kluwer Academic Publishers, 1998.

[20] Y. C. Baek and K. Koh, "Real-time scheduling of non-preemptive periodic tasks for continuous media retrieval," in *Proceedings of IEEE Region 10's Ninth Annual International Conference*, (New York, NY, USA), pp. 481–5, IEEE, 1994.

[21] S. Dolev and A. Keizelman, "Non-preemptive real-time scheduling of multimedia tasks," *Real-Time Systems*, vol. 17, pp. 23–39, 1999.

[22] S. Baruah, G. Buttazzo, S. Gorinsky, and G. Lipari, "Scheduling periodic task systems to minimize output jitter," in *Proceedings Sixth International Conference on Real-Time Computing Systems and Applications*, (Los Alamitos, CA, USA), pp. 62–9, IEEE, 1999.

[23] T. Kim, H. Shin, and N. Chang, "Deadline assignment to reduce output jitter of real-time tasks," in *Proceedings of the 16th Workshop on Distributed Computer Control Systems*, (Sydney, Australia), pp. 67–72, 2000.

[24] M. Di-Natale and J. A. Stankovic, "Scheduling distributed real-time tasks with minimum jitter," *IEEE Transactions on Computers*, vol. 49, no. 4, pp. 303–316, 2000.

[25] S. T. Cheng and A. K. Agrawala, "Allocation and scheduling of real-time periodic tasks with relative timing constraints," in *Proceedings Second International Workshop on Real-Time Computing Systems and Applications*, (Los Alamitos, CA, USA), pp. 210–217, IEEE, 1995.

[26] S. Acharya, R. Alonso, M. Franklin, and S. Zdonik, "Broadcast disks: Data management for asymmetric communications environments," in *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, pp. 199–210, 1995.

[27] A. Bar-Noy, R. Bhatia, J. Naor, and B. Schieber, "Minimizing service and operation cost of periodic scheduling," in *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 11–20, 1998.

[28] A. Bar-Noy, A. Nisgav, and B. Patt-Shamir, "Nearly optimal perfectly-periodic schedules," in *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pp. 107–116, 2001.

[29] A. Bar-Noy, V. Dreizin, and B. Patt-Shamir, "Efficient periodic scheduling by trees," in *Proceedings of IEEE INFOCOM*, (New-York, NY, USA), pp. 791–800, IEEE, 2002.

[30] C. Kenyon, N. Schabanel, and N. Young, "Polynomial-time approximation scheme for data broadcast," in *Proceedings 32nd STOC*, pp. 659–666, 2000.

[31] Z. Brakerski, A. Nisgav, and B. Patt-Shamir, "General perfectly periodic scheduling," in *Proceedings 21st ACM Symposium on Principles of Distributed Computing PODC'02*, (Monterey, California), 2002.

[32] H. Zhang, "Service disciplines for guaranteed performance service in packet-switching networks," *Proceedings of the IEEE*, vol. 83, pp. 1374–96, October 1995.

[33] A. K. Pnrekh and R. G. Gallager, "A generalized processor sharing approach to flow control - the single node case," in *Proceeding of IEEE INFOCOM*, pp. 915–924, 1992.

[34] A. Demers, S. Keshav, and S. Shenkar, "Analysis and simulation of a fair queueing algorithm," *Inrenetworking: Research and Experience*, vol. 1, pp. 3–26, 1990.

[35] J. C. Bennett and H. Zhang, "WF^{2}Q: worst-case fair weighted fair queueing," in *Proceedings of IEEE INFOCOM*, (Los Alamitos, CA, USA), pp. 120–128, IEEE, 1996.

[36] S. Golestain, "A self-clocked fair queueing schem for broadband applications," in *Proceeding of the IEEE INFOCOM*, (Toronto, Canada), pp. 636–646, 1994.

[37] D. Stiliadis and A. Varma, "Efficient fair queueing algorithms for packet-switched networks," *IEEE/ACM Transactions on networking*, vol. 6, pp. 175–185, April 1998.

[38] J. B. Nagle, "On packet switches with infinite storage," *IEEE Transactions on Communications*, vol. COM-35, no. 4, pp. 435–438, 1987.

[39] M. Katevenis, S. Sidiropoulos, and C. Courcoubetis, "Weighted round-robin cell multiplexing in a general-purpose ATM switch chip," *IEEE Journal on Selected Areas in Communications*, vol. 9, pp. 1265–79, October 1991.

[40] M. Shreedhar and G. Varghese, "Efficient fair queuing using deficit round-robin," *IEEE/ACM Transactions on networking*, vol. 4, pp. 375–385, June 1996.

[41] S. S. Kanhere, H. Sethu, and A. B. Parekh, "Fair and efficient packet scheduling using elastic round robin," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 324–336, 2002.

[42] D. Ferrari and D. C. Verma, "A scheme for real-time channel establishment in wide-area networks," *IEEE Journal on Selected Areas in Communications*, vol. 8, no. 3, pp. 386–379, 1990.

[43] F. M. Chiussi and V. Sivaraman, "Achieving high utilization in guaranteed services networks using early-deadline-first scheduling," in *Proceedings of the 6th International Workshop on Quality of Service (IWQoS)*, (New York, NY, USA), pp. 209–217, IEEE, 1998.

[44] J. Liebeherr, D. E. Wrege, and D. Ferrari, "Exact admission control for networks with a bounded delay service," *IEEE/ACM Transactions on Networking*, vol. 4, pp. 885–901, December 1996.

[45] L. Georgiadis, R. Guerin, and A. Parekh, "Optimal multiplexing on a single link: delay and buffer requirements," *IEEE Transactions on Information Theory*, vol. 43, no. 5, pp. 1518–35, 1997.

[46] A. Francini and F. M. Chiussi, "A weighted fair queueing scheduler with decoupled bandwidth and delay guarantees for the support of voice traffic," in *proceedings of IEEE Global Telecommunications Conference (GLOBECOM)*, (Piscataway, NJ, USA), pp. 1821–1827, 2001.

[47] Y. S. Sun, T. Yung-Cheng, and S. Wei-Kuan, "Frame-based worst-case weighted fair queueing with jitter control," *IEICE Transactions on Communications*, vol. E84-B, pp. 2266–78, August 2001.

[48] C. Jian, Q. Ying, C. Huimin, L. Yingchun, and C. Jainqiang, “Dynamic bandwidth allocation scheme for MCNS DOCSIS,” *Journal of Shanghai University*, vol. 2, pp. 328–333, December 1998.

[49] M. Droubi, N. Idirene, and C. Chen, “Dynamic bandwidth allocation for the HFC DOCSIS MAC protocol,” in *Proceedings Ninth International Conference on Computer Communications and Networks*, (Piscataway NJ USA), pp. 54–60, Springer-Verlag Berlin Germany, 2000.

[50] R. Rabbat and K. Y. Siu, “QoS support for integrated services over CATV,” *IEEE Communication Magazine*, vol. 37, pp. 64–68, January 1999.

[51] V. Sdralia, C. Smythe, P. Tzerefos, and S. Cvetkovic, “Performance characterization of the MCNS DOCSIS 1.0 CATV protocol with prioritized first come first served scheduling,” *IEEE Transactions on Broadcasting*, vol. 45, pp. 196–205, June 1999.

[52] P. Tzerefos, V. Sdralia, C. Smythe, and S. Cvetkovic, “Delivery of low bit rate isochronous streams over the DOCSIS 1.0 cable television protocol,” *IEEE Transactions on Broadcasting*, vol. 45, pp. 206–214, June 1999.

[53] W. M. Yin, C. J. Wu, and Y. D. Lin, “Two-phase minislot scheduling algorithm for HFC QoS services provisioning,” in *Proceedings of IEEE Global Telecommunications Conference GLOBECOM’01*, (Piscataway, NJ, USA), pp. 410–414, 2001.

[54] M. Hawa and D. W. Petr, “Quality of service scheduling in cable and broadband wireless access systems,” in *Proceedings 1ost IEEE International Workshop on Quality of Service*, pp. 147–255, IEEE, 2002.

[55] D. Bushmitch, S. Mukherjee, S. Narayanan, M. Ratty, and S. Q. Beser, “Supporting MPEG video transport on DOCSIS-compliant cable networks,” *IEEE Journal on Selected Areas in Communications*, vol. 18, pp. 1581–1596, September 2000.

[56] D. Jianghong, X. Zhongyang, C. Hao, and D. Hui, “Scheduling algorithm for MPEG-2 TS multiplexers in CATV networks,” *IEEE Transactions on Broadcasting*, vol. 46, pp. 249–255, December 2000.

[57] W. S. Lai, “Upstream bandwidth allocation for packet telephony in hybrid fiber-coax systems,” in *Proceedings of the 2000 Symposium on Performance Evaluation of Computer and Telecommunication Systems*, (San Diego, CA, USA), pp. 96–100, 2000.

[58] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness.* San Francisco: W H Freeman and Co., 1979.

[59] H. Kellerer, R. Mansini, and M. G. Speranza, “Two linear approximation algorithms for the subset-sum problem,” *European Journal of Operational Research*, vol. 120, pp. 289–296, January 2000.

[60] H. Kellerer, U. Pferschy, and M. G. Speranza, “An efficient approximation scheme for the subset-sum problem,” in *Algorithms and Computation, 8th International Symposium ISAAC ’97 Proceedings*, vol. xv+426, pp. 394–403, Springer-Verla Berlin Germany, 1997.

[61] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations.* England: John Wiley and Sons Ltd., 1990.

[62] E. G. Coffman Jr., M. R. Garey, and D. S. Johnson, “Approximation algorithms for bin packing: A survey,” in *Approximation Algorithms for NP-Hard Problems* (D. Hochbaum, ed.), pp. 46–93, Boston: PSW, 1996.

[63] D. Pisinger and P. Toth, “Knapsack problems,” in *Handbook of Combinatorial Optimization, vol. 1* (D. Z. Du and P. Pardalos, eds.), pp. 299–428, Kluwer Academic Publishers, 1998.

[64] D. S. Johnson, *Near-Optimal Bin Packing Algorithms.* Doctoral dissertation, Mathematics, Massachusetts Institute of Technology, Cambridge, MA, 1973.

[65] R. L. Graham, “Bounds on certain multiprocessing anomalies,” *Bell System Technical Journal*, vol. 45, pp. 1563–1581, 1966.

[66] R. L. Graham, “Bounds on multiprocessing anomalies,” *SIAM J. on Applied Mathematics*, vol. 17, pp. 263–269, 1969.

[67] N. Naaman and R. Rom, “Packet scheduling with fragmentation,” in *Proceeding of the 21st IEEE INFOCOM*, (Ney York, USA), pp. 427–436, 2002.

[68] N. Menakerman (Naaman) and R. Rom, “Analysis of bounded space bin packing algorithms,” Tech. Rep. 340, Technion EE publication CCIT, March 2001.

[69] Y. Azar, A. Broder, and A. Karlin, “On-line load balancing,” in *Proceedings 33^{rd} Annual Symposium on Foundations of Computer Science*, pp. 218–225, 1992.