Scheduling Data Transfers in Space Missions with Priorities and Interruptions: Technical Report

Julien Rouzot, C Artigues, Philippe Garnier, E Hebrard, Pierre Lopez, A Maillard, G Rabideau

▶ **To cite this version:**

Julien Rouzot, C Artigues, Philippe Garnier, E Hebrard, Pierre Lopez, et al.. Scheduling Data Transfers in Space Missions with Priorities and Interruptions: Technical Report. LAAS - CNRS; IRAP. 2025. ⟨hal-05264958⟩

HAL Id: hal-05264958
https://hal.science/hal-05264958v1

Submitted on 17 Sep 2025

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

Scheduling Data Transfers in Space Missions with Priorities and Interruptions: Technical Report

J. Rouzot*†, C. Artigues*, P. Garnier†, E. Hebrard*, P. Lopez*, A. Maillard‡, G. Rabideau‡
*LAAS-CNRS, Toulouse, France
†IRAP, Université de Toulouse, Toulouse, France
‡Jet Propulsion Laboratory, Pasadena, United States
julien.rouzot@laas.fr, artigues@laas.fr, hebrard@laas.fr, philippe.garnier@irap.omp.eu,
lopez@laas.fr, adrien.maillard@jpl.nasa.gov, gregg.rabideau@jpl.nasa.gov

***Abstract*—In deep space missions, scientific data generated by onboard instruments must be temporarily stored in local memory buffers before being downlinked to Earth during limited communication windows. Efficient scheduling of these data transfers is essential to prevent buffer overflows and data loss, particularly in the presence of uncertainties. Previous work has considered the overlapping Memory Dumping Problem (oMDP), which consists in assigning transfer priorities to the memory buffers and minimize the peaks memory usage, which reduces the risk of overflow. In this paper, we consider additional decisions in the transfer plans that are implementable in practice: data transfer from each buffer can be interrupted after a given time, once per downlink window, preventing it from dumping data until the next window, but redistributing the unused bandwidth to the other buffers. The new problem is called oMDPi (oMDP with interruptions). We obtain new complexity results, showing that oMDPi is NP-complete for at least two windows. While the complexity status of the single window oMDPi remains open, we propose a polynomial-time heuristic to solve it. We propose a hybrid heuristic to solve the general oMDPi, embedding a flow relaxation and a single window heuristic. The results on both real and realistic generated instances show that our heuristic achieves a significant reduction of memory peaks in a reasonable time compared to previous works, making the new policy attractive for future space missions.**

***Index Terms*—Scheduling, Data Transfers, Heuristics**

I. Introduction

Deep space exploration gives rise to combinatorial optimization problems, among which scheduling the activities of the science campaign takes a central part [1]–[3]. Another crucial issue is scheduling the transmission of data acquired by the spacecraft to Earth, facing a conjunction of complicating factors such as long communication times, limited bandwidth, intermittent visibility and ever-increasing data volumes to be transferred, due to multiple embedded instruments with increasingly sophisticated technology [4]. In a typical deep space exploration spacecraft, such as the ESA's Rosetta mission from which our work is issued, acquired data are stored in memory buffers attached to the instruments. Dumping these memory buffers to Earth can only be performed during spacecraft/Earth visibility periods defined by a finite set of downlink windows. The problem is simplified in some space missions, such as the Mars Express mission, where data acquisition and memory dumping cannot occur simultaneously [5]. For other missions such as Rosetta, data acquisition and data dumping tasks may overlap in time, and the resulting data transfer scheduling problem is called the overlapping Memory Dumping Problem (oMDP) [6]. During each downlink window, bandwidth is shared according to the priority assigned to each buffer, the priority list inside each downlink window being a decision variable. When communication is available, the buffers with the best priority share the bandwidth equally—unless one or more are empty, in which case the unused bandwidth is reallocated to the next priority level once all top-priority buffers are empty. Considering a given memory fill rate function for each buffer, the goal of oMDP is to assign dumping priorities to the buffers while avoiding overflows and minimizing the highest memory peak. Indeed, there exists some uncertainty about actual data production rates and on the bandwidth available to downlink the data. Reducing the peak usage therefore makes the plan more robust and reduces the risk of overflow.

In [6], the possibility to interrupt a buffer after a given time, once per downlink window—preventing it from dumping data until the next window—was presented as an additional decision, likely to provide additional flexibility that can be exploited to further reduce memory peak heights. During the Rosetta mission, this *stop-dump* command was tied to a specific buffer and a timestamp. Although the exact reason remains unclear, the European Space Agency (ESA) enforced a hard constraint: buffers could not be reactivated within the same window or interrupted multiple times. This restriction was likely motivated by operational concerns, such as minimizing the number of commands to reduce the risk of spacecraft failure. Due to the added complexity of jointly managing priority settings and interruption decisions, this promising research direction was left unexplored in [6]. In this paper, we include such decisions and introduce the overlapping Memory Dumping Problem with interruptions (oMDPi). We study its computational properties and propose heuristic solutions that make this additional flexibility practically exploitable. More specifically, we combine a flow-based LP model to compute *handover targets*—i.e. desired memory levels to be reached at the end of each downlink window—with a single window heuristic algorithm that determines a priority assignment to meet these targets without exceeding buffer capacities. Our source code, instances, experimental results and an extended

complexity analysis are available in our Git repository[1].

We define the oMDPi in Section II and present related works in Section III. We prove that oMDPi is NP-complete for two or more downlink windows in Section IV, and show that the problem with multiple windows and fixed priorities (oMDPi-FP) is also NP-complete. However, we show that, rather counterintuitively, optimal stop times may lead to local bandwidth loss. In Section V, we introduce a polynomial algorithm to compute the earliest interruptions for a given downlink window and priority assignment and we show that stopping the buffer as soon as possible is optimal in that case, thus proving that oMDPi-FP with a single window is not NP-hard. The complexity of oMDPi (without fixed priorities) for a single window remains open despite our efforts. We present a heuristic algorithm to find both the priority and interruptions times for a single window in Section VI. To deal with the multiple windows case, we propose a hybrid method using a Linear Program, presented in Section VII, to solve a flow-based relaxation of the problem to compute target memory levels over multiple windows. Our single window heuristic is used as a subroutine to fix priorities and interruption times targeting the memory levels. This multiple windows heuristic is described in Section VIII. We compare it to the best oMDP heuristics on both real and realistic randomly generated instances in Section IX. The results show that our method enables us to achieve a significant reduction of memory peaks in a reasonable time, making the new policy attractive for future space missions. We conclude and highlight research directions in Section X.

II. Problem Definition and Illustrative Example

In the oMDP, we have a set of buffers $\mathcal{B} = \{1, \ldots, n\}$. For each buffer $i \in \mathcal{B}$, we want to monitor the quantity $U_i(t) : \mathbb{R}^+ \mapsto \mathbb{R}^+$ of data that it holds between $t = 0$ and $t = h$, where h is the time horizon. In particular, we want to ensure that the memory level is bounded by the buffer capacity C_i. The buffers can be dumped only during a set of time windows in $\mathcal{W} = \{1, \ldots, m\}$, that start at w_j^{start} and end at w_j^{end}. The dump rate—i.e. bandwidth—available at such downlink window j is denoted δ_j. The memory peak $r_{i,j}$ for window j, is the ratio of the highest memory level of buffer i during the downlink window j and its capacity C_i. In the oMDP, the objective is to minimize the highest memory peak: $\min\ rmax = \max_{i \in \mathcal{B}, j \in \mathcal{W}} (r_{i,j})$.

We assume that the fill rate over time, denoted $f_i : \mathbb{R}^+ \mapsto \mathbb{R}^+$ for each buffer i, is entirely determined by the science observation plan and follows a piecewise constant function, where each nonzero segment corresponds to a distinct observation.

Transfers can be controlled by assigning priorities to buffers, which can be modified only at the start of each downlink window. More precisely, for each buffer i, priorities $p_{i,j}$, are within the domain $[1, \ldots, n]$, where 1 is the *best* priority and n the *worst*. The transfer policy follows a simple priority-based Round-Robin scheme for sending atomic data packets. More precisely, whenever bandwidth is available, the system transfers a single data packet from one of the non-empty buffers with the highest current priority. If all such buffers are empty, it proceeds to the next priority pool, continuing this process until a non-empty buffer is found or all priority pools have been checked. Since multiple buffers can share the same priority, ties are broken using a circular queue, which means that among buffers with equal priority, preference is given to the one that transferred data least recently.

While the procedure for downlinking data according to the priorities is discrete, it has been shown in [2] that the transfer rate functions resulting from a priority policy p can be approximated as piecewise linear functions with a very minimal loss of precision, as the size of the data block are extremely small in front of the fill rates and the dump rates. In [7], a procedure based on the work presented in [2] is proposed to compute the transfer rates and the memory level functions resulting from the fill rate functions and a priority assignment, assuming that buffers are filled and dumped continuously.

Definition II.1 (Transfer Rates [7])**.** Given a downlink window j and a priority assignment p, the transfer rate functions $g_i^p(t)$ can be computed recursively by considering the buffers from the best priority rank to the worst. Let $\Omega = \{i_1, \ldots, i_k\}$ be the set of buffers with best priority, ordered by ascending memory level, breaking ties with ascending fill rate at time t. The available bandwidth Δ_j for buffers in Ω, is initially set to δ_j. Then the transfer rate $g_{i_1}^p(t)$ of buffer i_1 at time t is $\Delta_j/|\Omega|$ if its memory is not empty (an equal share of the bandwidth) and $\min(f_{i_1}(t), \Delta_j/|\Omega|)$ otherwise (as fast as it is filled, but no more than an equal share). Then for $\ell \in [2, k]$ the transfer rate $g_{i_\ell}^p(t)$ is computed similarly, after setting Δ_j to $\Delta_j - g_{i_{\ell-1}}^p(t)$ and Ω to $\Omega \setminus \{i_{\ell-1}\}$. Finally, if $\Delta_j > 0$, the transfer rates are computed in the same way with the buffers in the next priority ranks (one rank at a time), until either the residual bandwidth is null or all buffers have been considered.

Definition II.2 (Memory Level)**.** Given the fill rate and transfer rate functions f_i and g_i^p, and the initial memory level for each buffer $U_i(0)$, the memory level over time for buffer i, $U_i^p(t)$, is defined as the integral, over time, of the data production minus the data transfer:

$$U_i^p(t) = U_i(0) + \int_0^t \left(f_i(t') - g_i^p(t')\right)\, dt'$$

Definition II.3 (oMDP)**.** Given the buffer set $\mathcal{B}$, the downlink windows $\mathcal{W}$, the fill rate functions f_i, and the initial memory level $U_i(0)$ for each buffer i, what is the priority assignment p that minimizes:

$$rmax = \max_{i \in \mathcal{B}} \left(\max_{t \in [0,h]} \left(U_i^p(t)\right) / C_i \right)$$

The oMDPi is similar to oMDP except that besides priorities, a unique *interruption* time $s_{i,j} \in [w_j^{start}, w_j^{end}]$ can be assigned to every buffer i for each downlink window j. When

[1] https://gitlab.laas.fr/roc/julien-rouzot/omdpi

the interruption time of a given buffer is reached, this buffer is ignored in the Round-Robin (i.e. it completely stops dumping until the next window).

In this new setup, the transfer rate functions are calculated as explained in definition II.1, except that, given a buffer i and a downlink window j, if $t \geq s_{i,j}$, then $g_i^{p,s} = 0$ and i is removed from Ω. As in oMDP, the resulting memory level over time for each buffer i is:

$$U_i^{p,s}(t) = U_i(0) + \int_0^t \left(f_i(t') - g_i^{p,s}(t') \right) \, dt'$$

Definition II.4 (oMDPi)**.** Given the buffer set $\mathcal{B}$, the downlink windows $\mathcal{W}$, the fill rate functions $f_i(t)_{t\in[0,h]}$ and the initial memory level $U_i(0)$ for each buffer i, what is the priority and interruptions assignment $\{p, s\}$ that minimizes:

$$rmax = \max_{i\in\mathcal{B}} \left(\max \left(U_i^{p,s}(t)_{t\in[0,h]} \right) \right)$$

Definition II.5 (D-oMDPi)**.** Given the buffer set $\mathcal{B}$, the downlink windows $\mathcal{W}$, the fill rate functions $f_i(t)_{t\in[0,h]}$, the initial memory level $U_i(0)$ for each buffer i and a threshold v, is there a priority and interruptions assignment $\{p, s\}$ such that:

$$rmax = \max_{i\in\mathcal{B}} \left(\max \left(U_i^{p,s}(t)_{t\in[0,h]} \right) \right) \leq v$$

Figure 1 provides an illustrative example of a solution of the oMDPi. Here, we consider an instance with $n = 3, m = 2, C_1 = C_2 = C_3 = 4$. Initially, the buffers are empty. The downlink windows are represented in light green, with dump rates $\delta_1 = 2$ and $\delta_2 = 3$. The white areas correspond to non-visibility windows ($\delta = 0$). Colored rectangles represent observations (height = fill rate, length = duration). The red curves show the memory usage functions $U_i(t)$, resulting from priority and interruption decisions.

For the first downlink window, $p_{1,1} = 1$ and $p_{2,1} = p_{3,1} = 2$, so buffer 1 has priority. At $t = 0$, buffer 1 produces 2 data units that cannot be dumped. Since buffer 1 has priority, we have $g_1(t) = 2$ between $t = 1$ and $t = 3$, after which it becomes empty. Then buffers 2 and 3 share the residual bandwidth equally. In the second downlink window, interruptions (depicted as stars in the figure) are introduced. At $t = 6$, all buffers contain 2 units of data and they all have a high production event at different parts of the next downlink window. The priorities are $p_{1,2} = 1$, $p_{2,2} = 2$, $p_{3,2} = 3$. We stop buffer 1 at $t = 7$, giving it priority until then, after which buffer 2 has priority. Stopping buffer 2 at $t = 8$ gives buffer 3 priority until the end of the window. In this example, $rmax = 50\%$ is optimal, as a peak of at least 50% occurs for buffer 1 during the non-visibility period between downlink window 1 and 2.

III. Related Work

The first approach for scheduling data transfers in deep space missions with separate memory buffers was introduced in the context of the Mars Express mission. In [5], the Mars Express Memory Dumping Problem (Mex-MDP) was

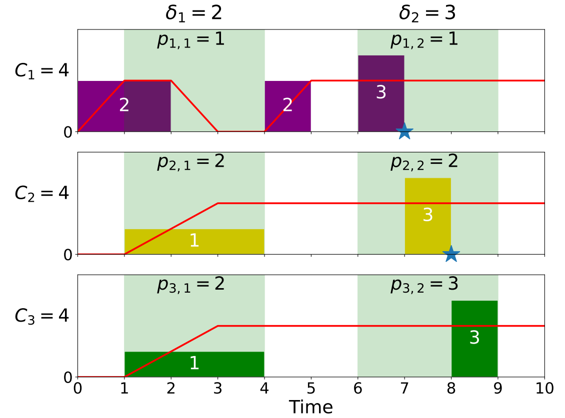

Fig. 1: Illustrative example of an oMDPi solution with three buffers and two downlink windows.

formulated, wherein data production and data transfers were treated as distinct processes: data are generated during non-visibility periods and subsequently transmitted during visibility windows. The authors expressed the problem as a CSP, and presented a two-step optimization procedure based on a greedy solver and a Tabu search algorithm to solve it.

In other missions, data production and transfer can occur simultaneously. In [6], [8] the authors introduced oMDP for the Rosetta mission and presented the fast Downlink Count heuristic to solve the problem. This first heuristic approach for oMDP is based on the following assumption: buffers that overflow first if no data dump is performed should be given a better priority. More precisely, for each downlink window, Downlink Count *counts* the number of windows before an overflow occurs, if $\delta_j = 0$, $j \in \{1, \ldots, m\}$. Then, the priorities are distributed to the buffers, starting with the best priority for the buffer with the lowest downlink count. In the case of ties, the same priority is assigned. This process is repeated until all priorities are assigned. This heuristic has a very low computing cost as we only need to sum the fill rates events until the overflow is reached, for each buffer, for a given downlink window. This simple heuristic yields good solutions on its own and was used by Rabideau et al. in their Iterative Leveling method, consisting of iteratively reducing the overflow limit.

Even though the Rosetta mission has ended, efficient algorithms for scheduling data transfers via priority assignments remain valuable for future missions. The oMDP has been further studied in [7], where the authors introduced a fast and exact Simulation procedure to compute memory usage functions based on a given priority assignment. They also proposed a polynomial-time algorithm, SingleWindow, to determine the optimal priority assignment for a single downlink window, as well as a Large Neighborhood Search-based heuristic (RepairDescent) for solving the multiple-window problem. In this paper, we build on this work, so let us present the Simulation and SingleWindow algorithms.

In the SIMULATION algorithm, a first important step is to create a list of events, including the start or the end of fill rate and dump rate events, ordered by time. These events are evaluated chronologically to update the fill rates and compute the transfer rates resulting from the current memory usage, fill rates, dump rate and priority assignment p. Given a downlink window j and a priority assignment p, the transfer rate at time t is computed recursively by considering the buffers from the best priority rank to the worst. Let $\Omega = \{i_1, \ldots, i_k\}$ be the set of buffers with best priority, ordered by ascending memory usage, breaking ties with ascending fill rate at time t. The available bandwidth Δ_j for buffers in Ω, is initially set to δ_j. Then the transfer rate $g^p_{i_1}(t)$ of buffer i_1 at time t is $\Delta_j/|\Omega|$ if its memory is not empty (an equal share of the bandwidth) and $\min(f_{i_1}(t), \Delta_j/|\Omega|)$ otherwise (as fast as it is filled, but no more than an equal share). Then for $\ell \in [2, k]$ the transfer rate $g^p_{i_\ell}(t)$ is computed similarly, after setting Δ_j to $\Delta_j - g^p_{i_{\ell-1}}(t)$ and Ω to $\Omega \setminus \{i_{\ell-1}\}$. Finally, if $\Delta_j > 0$, the transfer rates are computed in the same way with the buffers in the next priority ranks (one rank at a time), until either the residual bandwidth is null or all buffers have been considered. For a given downlink window, these transfer rates do not change unless an observation starts or ends—i.e. a fill rate changes—or a buffer becomes empty. When such an event occurs, all transfer rates must be recomputed, as the buffer becoming empty will redistribute its share of the bandwidth. In the case of a buffer becoming empty, a new event is inserted if it occurs before the next breakpoint. This process is repeated until the time horizon is reached, and computing the resulting memory usage functions is straightforward when the transfer rates are known. With k the number of observations, this algorithm simulates a priority assignment and returns the memory usage functions for each buffer in $O(k \log k + n^2 k \log n)$.

In the SINGLEWINDOW algorithm, the SIMULATION procedure is used iteratively to eliminate invalid priority values for a given downlink window j and a $rmax$ limit until a valid priority assignment is found. Priorities are initially set to n, the worst priority. After running SIMULATION, any overflowing buffer must be assigned a better priority to satisfy the $rmax$ constraint. The priorities are then updated and the process repeats until a solution with no overflow is found, or no further improvement in priorities is possible. SINGLEWINDOW algorithm requires $O(n^2)$ calls to SIMULATION. By performing a dichotomic search on $rmax$, the optimal solution for a single downlink window can be determined in polynomial time.

IV. PROBLEM COMPLEXITY AND PROPERTIES

While oMDP has been proven NP-hard in [7], the introduction of the interruption variables makes our problem a relaxation of oMDP of a-priori unknown complexity.

Theorem IV.1. *D-oMDPi is NP-complete, for at least two downlink windows and even if priorities are fixed.*

Proof. (Sketch) Here, we only give a sketch to help the reader understand the full proof that follows. We prove the theorem by reduction from PARTITION Problem [9], which is a special case of the SUBSETSUM problem. The decision variant of the PARTITION problem ask whether there exists an even partition of a set $S = \{a_i \mid 1 \leq i \leq n\}$, with $\sum_{i=1}^n a_i = 1$, such that there exists a subset $I \subseteq \{1, \ldots, n\}$ for which $\sum_{i \in I} a_i = \sum_{i \notin I} a_i$ (or equivalently $\sum_{i \in I} a_i = \frac{1}{2}$).

We use a first downlink window in which the set I of buffers that are interrupted represents the first half of the partition ($i \in S$ iff a_i in the first half). The same downlink window forces, in order to meet capacity constraints, to stop enough buffers so that $\sum_{i \in I} a_i \geq \frac{1}{2}$. Moreover, at the end of this window, a small amount of data will be produced in each buffer i, proportional a_i. This quantity will only remain on the interrupted buffers (those in S) and will be dumped by other buffers. We use a second downlink window that is feasible if and only if the residual data is small enough, which is true if $\sum_{i \in I} a_i \leq \frac{1}{2}$. Therefore, there exists a feasible plan for D-oMDPi if and only if there exists an even partition.

Moreover, in the construction, we use gadgets to force a single feasible priority assignment. Therefore, the theorem still stands when priorities are given and when there are no more than two downlink windows.

■

Now that we provided insights for the proof, the complete demonstration is described below:

Proof. D-oMDPi is in NP because the SIMULATIONSTOP algorithm (see Section V) can check a priority affectation and an interruption plan in polynomial time.

We give a construction π which is a polynomial reduction from an instance x of PARTITION to an instance $\pi(x)$ of D-oMDPi. We assume (without loss of generality) that the instance x of PARTITION is a set of real numbers $A = \{a_1, \ldots, a_n\}$ such that $\sum_{i=1}^n a_i = 1$, and hence it is satisfiable if and only if there exists $I \subseteq \{1, \ldots, n\}$ such that $\sum_{i \in I} a_i = \frac{1}{2}$.

In the reduction, we use $n+1$ buffers and 2 windows. The capacity of buffer $1 \leq i \leq n$ is $a_i(1 + \frac{1}{m})$ with m a large number (at least so that $\min_i(a_i)/m < 1$). The capacity of buffer $n+1$ is another large number L (we can use $L = \max(2, n)$). All buffers are empty at the start of the first downlink window. The first downlink window has dump rate $\delta_1 = n$ and can be divided into 5 parts of length 1, starting at $t = 0$, followed by a part of duration L/n.

- In the first part, buffers $1 \leq i \leq n$ produce $1 + a_i + \frac{a_i}{m}$ and buffer $n+1$ produces $n - 1 - \frac{1}{m}$.
- In the second part, only buffer $n+1$ produces $L - \frac{1}{2}$.
- In the third part, the fill rate of buffer $n+1$ is null, and the fill rate of buffer $1 \leq i \leq n$ is $1 + a_i$.
- In the fourth part, the fill rate of buffer $n+1$ is n, and the fill rate of buffer $1 \leq i \leq n$ is null.
- In the fifth part, the fill rate of buffer $n+1$ is null, and the fill rate of buffer $1 \leq i \leq n$ is $\frac{a_i}{m}$.
- In the last part, the fill rate is null for all buffers.

The second downlink window has a dump rate $\delta_2 = n+1$. It is divided into 5 parts: the first, third and fifth of duration 1 and the second and fourth of duration $L/(n+1)$.

- In the first part, the fill rate of buffer $n+1$ is $L+n+1-\frac{1}{2}-\frac{1}{2m}$, and the fill rate of all other buffers is null.
- In the second part, the fill rate is null for all buffers.
- In the third part, the fill rate of buffer $n+1$ is $L+1$ and for $1 \le i \le n$, the fill rate of buffer i is $1+a_i+\frac{a_i}{m}$.
- The fourth and fifth parts are the same as the second and third, respectively.

We now prove that x is a satisfiable PARTITION instance if and only if $\pi(x)$ is a satisfiable D-oMDPi instance. The following lemmas are steps to prove Lemma IV.9 which is the "if" part of the proof. Lemma IV.10 is the "only if" part.

Lemma IV.2. *In the first downlink window, buffers $1 \le i \le n$ must have equal priority, and buffer $n+1$ must have a strictly worse priority.*

Proof. Every buffer produces more than its capacity during the first downlink after the first part. Therefore, none of them can be interrupted before $t = 1$. Moreover, since the total production of buffers $i \in \{1, \ldots, n\}$ on the first part is equal to the sum of the maximum that can be dumped (n) and of their total capacities ($a_i + \frac{a_i}{m}$), they must use all the bandwidth, hence buffer $n+1$ must be of strictly worse priority. Finally, since every buffer $1 \le i \le n$ must dump exactly 1, they must all have the same priority. ■

Lemma IV.3. *At the start of the third part of the first downlink, buffer $n+1$ contains $L - \frac{1}{2}$ data, and all other buffers are empty.*

Proof. Simple application of the problem definition since we know the priority allocation by Lemma IV.2, and there is no interruption possible because every buffer still has too much data to produce at this point. ■

Lemma IV.4. *Buffer $n+1$ must use all but $\frac{1}{2}$ of the bandwidth on the fourth part of the first downlink.*

Proof. By Lemma IV.2, other buffers use all the bandwidth on the third part and therefore its memory level is still $U = L - \frac{1}{2}$ at the start of the fourth part (the same as the level at the end of the second part by Lemma IV.3). Its production on this part is $P = n$ and its capacity is $C = L$. Let D the data dumped from buffer $n+1$ on the fourth part. We must have $U + P - D \le C$, i.e., $D \ge n - \frac{1}{2}$. Since the dump rate is n and the part duration is 1, at most $\frac{1}{2}$ can be dumped by other buffers. ■

Lemma IV.5. *A non-empty set $I \subseteq \{1, \ldots, n\}$ of buffers must be interrupted, and each interrupted buffer a_i cannot be interrupted strictly earlier than $t = 3$ in the first downlink.*

Proof. Buffers $1 \le i \le n$ all have priority over buffer $n+1$ and have a total memory level of 1. Therefore, if none of them is interrupted, they will transfer 1 over this part, which is a contradiction with Lemma IV.4. Moreover, since the memory level of buffer i is at most $a_i + \frac{a_i}{m}$, and since it dumped at rate 1, it cannot have been interrupted before time $t = 3$ since its memory level at this point is a_i and $\frac{a_i}{m}$ will be produced later on. ■

By Lemma IV.5 there is a non-empty set of buffers I that have to be interrupted during the fourth part. Let $0 < \alpha_i \le a_i$ be the quantity of data in the buffer at time $t = 4$ and *not* dumped in the fourth part.

Lemma IV.6. $\sum_{i \in I} \alpha_i \ge \frac{1}{2}$ *and* $\sum_{i \in I} a_i \ge \frac{1}{2}$

Proof. By Lemma IV.4, only $\frac{1}{2}$ data in total can be dumped by buffers $1 \le i \le n$ during part 4 of the first downlink. Since by Lemma IV.2 they all have the strictly better priority than buffer $n+1$ there would be a contradiction if $\sum_{i \in I} \alpha_i < \frac{1}{2}$. Since $\alpha_i \le a_i$, we have $\sum_{i \in I} a_i \ge \frac{1}{2}$. ■

Lemma IV.7. *At the end of the first downlink window:*

- *the memory level of buffer $i \in \{1, \ldots, n\} \setminus I$ is null;*
- *the memory level of buffer $i \in I$ is $\alpha_i + \frac{a_i}{m}$;*

Proof. The total amount of data to dump from buffers $1 \le i \le n$ in the second part is at most $1 + \frac{1}{m} < n$ and since they have priority over the remaining buffer $n+1$ by Lemma IV.2, all the data held on an uninterrupted buffer will be dumped. Now, buffer $i \in I$ has kept α_i of the data it had at time $t = 4$, and it is filled with an extra $\frac{a_1}{m}$ before time $t = 5$ (which it cannot dump since it is interrupted). ■

Lemma IV.8. *In the second downlink, all buffers have equal priority, and no interruption is possible before the fifth part.*

Proof. Each of the $n+1$ buffers produces its capacity plus one unit of data in the last part, and only $n+1$ units can be dumped. Hence they must all dump at least one unit and hence no interruption is possible before. The same observation is true in the third part, and since no interruption is possible in this part, the only feasible priority allocation is to have all buffers at the same priority level. Moreover, note that there is enough time in the second and fourth parts to empty all buffers. ■

Lemma IV.9. *If $\pi(x)$ is satisfiable, then x is satisfiable.*

Proof. In the second downlink window, buffers $i \in I$ begin with level $\alpha_i + \frac{a_i}{m}$, and all other buffers are empty because of the long inactive sixth part of the first downlink. The overall initial level of buffers $1 \le i \le n$ is less than $1 + \frac{1}{m}$. Since all buffers have the same priority by Lemma IV.8 a bandwidth of $n+1$ is sufficient for all buffers $1 \le i \le n$ to be empty at time $t = 1$ in the second downlink. Therefore, only buffer $n+1$ can have some non-null memory level at that point. At the start of the downlink, the total memory is $U \ge \frac{1}{2} + \sum_{i \in I} \frac{a_i}{m}$. The sum of the production and of the initial level minus the total data dump $n+1$ must be less than or equal to buffer $n+1$'s capacity, and therefore:

$$\frac{1}{2} + \sum_{i \in I} \frac{a_i}{m} + L + n + \frac{1}{2} - \frac{1}{2m} - (n+1) \le L$$

that is,

$$\sum_{i \in I} a_i \le \frac{1}{2}$$

By Lemma IV.6 we therefore have $\sum_{i \in I} a_i = \frac{1}{2}$, and hence I is a solution of the PARTITION problem. ■

Lemma IV.10. *If x is satisfiable, then $\pi(x)$ of is satisfiable.*

Proof. Only one priority assignment is allowed in our construction, so only the interruption decision matters. We simply interrupt buffer i at time $t = 3$ in the first downlink window and do no other interruptions. Then we have a solution of D-oMDPi with $\alpha_i = a_i$ for all $1 \leq i \leq n$. ■

The proof is complete as Lemma IV.9 and IV.10 show the equivalence of satisfiability of x and $\pi(x)$.

■

Theorem IV.11. *Bandwidth loss may be mandatory to find a solution for oMDPi.*

Proof. See Figure 2. ■

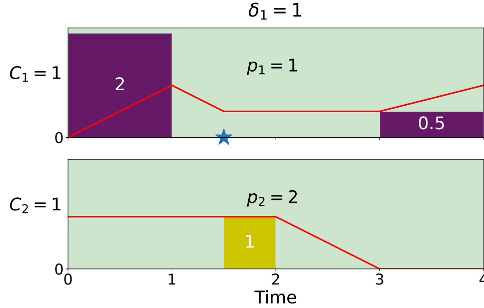

Fig. 2: Here, the capacity of both buffers is 1. At $t = 0$, buffer 1 holds 0 data unit and buffer 2 holds 1 data unit. We have $\delta_1 = 1$. In this example, buffer 1 must have strict priority over buffer 2, as it must transfer at least 1 between $t = 0$ and $t = 1$. Buffer 2 cannot be stopped, even with the best priority, before $t = 1.5$. If buffer 1 is not interrupted at this date, buffer 2 will exceed its capacity, as it must transfer 0.5 between $t = 1.5$ and $t = 2$. As a result, we can see that the bandwidth between $t = 3$ and $t = 4$ is lost, as buffer 1 is interrupted and buffer 2 is empty.

This shows that a trade-off between solution quality and bandwidth loss must be considered when solving oMDPi. In the remainder of this paper, we focus on minimizing the highest memory peak $rmax$, without considering the bandwidth under-use in the objective function, but we will quantify the bandwidth loss of solutions obtained in Section IX.

V. Earliest Interruptions with Fixed Priorities

In Section III, we reminded the SIMULATION algorithm introduced in [7]. This algorithm computes the memory evolution according to a priority assignment only, assuming that the buffers are never stopped. We extend this procedure to handle the interruption variables. Moreover, we present a procedure to compute the earliest interruption times, according to a set of *handover targets*, which corresponds to the memory state that we want to reach at the end of each downlink window.

Algorithm 1 SIMULATIONSTOP: Simulation with earliest stop time for all buffers given a priority assignment and handover targets. Return the index of the first buffer to overflow or 0.

Require: j, δ_j, w_j^{start} and w_j^{end}, $\mathcal{B}$, f_i, p_j, $U_i(start_j)$, the handover targets h_j^{tg}, a $rmax$ limit.

1: $t_{now} \leftarrow w_j^{start}$
2: $t_{next} \leftarrow w_j^{start}$
3: *events* $\leftarrow$ **MakeEventList**($f_i(t)$ over $[w_j^{start}, w_j^{end}]$)
4: $k \leftarrow 0$
5: **while** $t_{now} \leq end_j$ **do**
6: $\quad t_{now} \leftarrow t_{next}$
7: $\quad$ **while** $t_{now} = t_{next}$ **do**
8: $\quad\quad$ **if** *events*[k].*type* $=$ *stop* **then**
9: $\quad\quad\quad$ **StopBuffer**(*events*[k].*buffer*)
10: $\quad\quad$ **else**
11: $\quad\quad\quad$ **UpdateFillRate**(*events*[k].*buffer*, *events*[k].*value*)
12: $\quad\quad k \leftarrow k + 1$
13: $\quad\quad t_{next} \leftarrow$*events*[$k$].*time*
14: $\quad$ **ComputeTransferRates**(p_j)
15: $\quad$ **for** $i \in B$ **do**
16: $\quad\quad t_{empty} \leftarrow$ **EmptyTime**(i)
17: $\quad\quad$ **if** $t_{empty} < t_{next}$ **then**
18: $\quad\quad\quad t_{next} \leftarrow t_{empty}$
19: $\quad$ **for** $i \in B$ **do**
20: $\quad\quad t_{stop} \leftarrow$ **StopTime**(i, t_{next}, h_j^{tg})
21: $\quad\quad$ **if** $t_{stop} < t_{next}$ **then**
22: $\quad\quad\quad t_{next} \leftarrow t_{stop}$
23: $\quad\quad\quad$ **InsertStopEvent**(*events*, t_{stop}, i)
24: $\quad$ **if** $idx \leftarrow$ **GetOverflowIndex**(t_{next}, $rmax$) > 0 **then**
25: $\quad\quad$ **return** idx
26: $\quad$ **JumpTo**(t_{next})
27: **return** 0

First, it is straightforward to include fixed interruption times into the SIMULATION algorithm, as this only consists in adding new breakpoints, and setting the transfer rate of each buffer to 0 when their corresponding stop event is reached. But rather than providing exact interruption times, it can be useful to compute the earliest interruptions for the buffers according to a priority assignment p_j for the current downlink window j, a peak target $rmax$ and a list of handover targets $h_{i,j}^{tg}$. The handover targets are a memory state at the end of the current downlink window j, that is assumed to be advantageous to reach—this will be useful for multiple windows. This new algorithm is called SIMULATIONSTOP and is an extension of the SIMULATION algorithm. In this procedure, we consider a single downlink window where we simulate a priority assignment and compute the earliest interruptions to match the given handover oracle. We present it in detail in Algorithm 1.

In SIMULATIONSTOP, we have a list of events that are triplets (type, buffer, value) indicating whether the current event is an interruption or a fill rate change, the corresponding buffer, and the new fill rate value in case of fill rate events. Note that we are not considering dump rate change events, as we are only considering a single downlink window (multiple downlink windows can be simulated sequentially). We initialize this list with the breakpoints corresponding to the fill rate events of our instance. We keep this list sorted by ascending time. We track of the current and next time

points (lines 1, 2) and the current event (line 4). In a main loop, we evaluate every event, from the earliest to the latest. For each interruption event, we stop the corresponding buffer (line 9) and for each fill rate event, we update the fill rate (line 11). When all events at the current time point have been processed, we compute the actual transfer rates (line 14), with the procedure explained in Section III. Once the transfer rates are computed, we have to check whether a buffer becomes empty (line 16), or whether it can be interrupted before the next event (line 20). To find if a buffer can be stopped before the next event, we compute the amount of data d_i left to dump for each buffer i between the next time point t_{next} and the downlink window end time end_j, with j being the window index. We have $d_i = \int_{t_{next}}^{end_j} f_i(t)\,dt$. This amount of data can be pre-computed for each initial breakpoint (corresponding to the fill rate events), and can be computed in $O(1)$ for any other time point as the fill rate of a buffer is constant between two breakpoints. If $d_i > h^{tg}_{i,j}$, we cannot stop the buffer between the current time point t_{now} and t_{next}. Otherwise, we want to interrupt buffer i such as $U_i(t_{next}) = h^{tg}_{i,j} - d_i$. Thus, we can stop the buffer at time $t_{stop} = t_{now} + \left[U_i(t_{now}) - (h^{tg}_{i,j} - d_i) + f_i(t_{now}) \times (t_{next} - t_{now})\right] / g_i(t_{now})$ or $t_{stop} = t_{now} + (h^{tg}_{i,j} - d_i)/f_i(t_{now})$ if $g_i(t_{now}) = 0$. If a buffer can be stopped before t_{next}, we insert a new stop event with time t_{stop} (line 23). We then jump to the next time point, and we update the new memory state (line 26). This is repeated until all events have been processed or a buffer exceeds its $rmax$ target. In that case, we return the index of the first overflowing buffer—this will be useful to find both the priorities and the stop times in the single-window heuristic. Otherwise, we return 0.

Theorem V.1. *For a single downlink window, when priorities are fixed, stopping the buffers as soon as possible is optimal.*

Proof. For a single downlink window j, the handover targets h^{tg}_j can be set to $rmax \times C_i,\ \forall i \in \mathcal{B}$. Let t be the earliest time to safely stop buffer k, given a single downlink window j and a fixed priority assignment p_j, with respect to $h^{tg}_{k,j}$. The transfer rate of buffer k is always greater than or equal to 0 and becomes 0 when it is interrupted. If buffer k is stopped at t, the overall bandwidth available for the other buffers can only increase after t. Moreover, the bandwidth share allocated to any particular buffer cannot decrease when increasing the total bandwidth available. As the fill rates and priorities are fixed, increasing the transfer rates of the other buffers can only decrease their memory peaks. Therefore, stopping buffers as soon as possible minimizes the highest memory peak. ■

VI. Heuristic for oMDPi on a SingleWindow

Although we showed in Section IV that oMDPi is NP-complete for at least 2 downlink windows, the problem complexity for a single downlink window is left open. In this section, we propose a polynomial method close to the SingleWindow algorithm in [7], to build a solution with respect to a given $rmax$ threshold and a set of *handover constraints*, i.e. a memory limit for each buffer at the end of the downlink window. The handover *constraints* are different from the handover *targets* presented in Section V. The handover targets are used in the SimulationStop as a memory state to reach *if possible*. The handover constraints are more restrictive, as we consider that a buffer exceeding its handover constraint is overflowing. The motivation behind both concepts, as well as their roles in the broader heuristic for handling multiple downlink windows, will be discussed further in Section VIII. For now, we assume that the handovers are provided by an oracle. We present SingleWindowStop in Algorithm 2.

Algorithm 2 SingleWindowStop: Heuristic for a single downlink window, for given $rmax$ limit, h^{tg}_j and h^{ct}_j

```
Require: ins_j, h^tg_j, h^ct_j, rmax.
 1: p ← {n | ∀i ∈ B}
 2: while true do
 3:     O ← {}
 4:     k ← SimulationStop(ins_j, p, h^tg_j, rmax)
 5:     if k = 0 then
 6:         O ← {i | i ∈ B, U_i(end_j) > h^ct_{i,j}}
 7:         if |O| > 0 then
 8:             k ← RandomElement(O)
 9:     if k > 0 then
10:         p_k ← p_k − 1
11:         if p_k = 0 then
12:             return false
13:     else
14:         return true
```

In this heuristic, we use SimulationStop to successively try priority assignments until we find one that respects the $rmax$ threshold and handover constraints. For ease of notations, ins_j correspond to the fixed parameters of the instance to solve for the current downlink window, namely: $j, \delta_j, w^{start}_j, w^{end}_j, \mathcal{B}, f_i\,\forall i \in \mathcal{B}, U_i(w^{start}_j)\,\forall i \in \mathcal{B}, events$. After each failed simulation run, we have to find a new priority assignment to prevent all buffers from overflowing. Starting with priority $p_i = n,\ \forall i \in \mathcal{B}$ (i.e. worst priority for every buffer), we give the *first overflowing buffer only* a better priority when the simulation fails. By giving this buffer a better priority, we hope to prevent it from overflowing, but also to stop it earlier, thus releasing some bandwidth for the other buffers. First, we initialize all priorities to n, the worst priority level (line 1). We then run SimulationStop with the current priority assignment and handover targets (line 4). Unlike the SingleWindow algorithm of Hebrard et al. (see Section III), when buffers are overflowing, we only improve the priority of one buffer (line 10), instead of giving all overflowing buffers a better priority. Indeed, when adding interruption variables, a buffer overflowing with its current priority can be "fixed" if another buffer with a better priority is interrupted earlier. If no buffer exceeds the $rmax$ target, we check the handover constraints (line 6). If one or more buffer exceed those constraints, we randomly select one of them to increase its priority. We repeat this process until no better priority is available for the current overflowing buffer (line 12) or a valid solution is found (line 14). This satisfiability test can be used in a dichotomic search procedure to minimize $rmax$.

VII. LP RELAXATION

In this section, we present a Linear Programming model adapted from [5] that is a flow relaxation of oMDPi. For this LP, we forget the priorities and interruptions and assume that the bandwidth can be arbitrarily split between each breakpoint—those dynamically generated when a buffer is emptied are not considered. We still want to minimize the highest memory peak $rmax$. The main difference with [5] is the possibility of dumping buffers while they are producing, whereas the authors assumed the data to be produced only during non-visibility windows, which is a weaker relaxation.

Let $\mathcal{I} = \{1, \dots, l\}$ be the indices of the initial breakpoints, with l the number of these time points. Like in SIMULATION, we build the breakpoint set $\mathcal{T}$ by adding time events where either a fill rate or the dump rate changes—i.e. the time label of each event in the list $events$. If multiple changes occur at the same time, only one inflection time is added to $\mathcal{T}$. Thus, elements of $\mathcal{T}$ are unique and sorted in ascending order. We denote t_k the time of the k-th breakpoint in $\mathcal{T}$. The total bandwidth available and the total fill of buffer i between time t_k and t_{k+1} are b_k and $f_{i,k}$, respectively. To ensure that all available bandwidth is used, we compute the expected global memory state at the end of the last downlink window, u^{end}. We compute this global memory state by running a solution with priorities only ($(p_{i,j} = 1 \; \forall i \in \mathcal{B}, \forall j \in \mathcal{W})$, without capacity constraints. The sum of the memory state of each buffer at $t = w_m^{end}$ is equal to u^{end}. We introduce the following variables for our LP.

$$u_{i,k} \qquad i \in \mathcal{B}, k \in \mathcal{I} \qquad \textit{Memory state} \qquad (1)$$

$$tr_{i,k} \qquad i \in \mathcal{B}, k \in \mathcal{I} \setminus l \qquad \textit{Transfer rate} \qquad (2)$$

The memory variables $u_{i,k}$ store the memory state of buffer i at the k-th breakpoint. The variables $tr_{i,k}$ represent the bandwidth share allocated to buffer i between time points k and $k+1$. Our objective is to minimize the highest memory peak $rmax = \max(u_{i,k}/C_i)$. We linearize this objective by introducing a new variable α to minimize. By forcing all ratios between the memory variables and their capacity to be lower than or equal to α, minimizing α is equivalent to minimizing $rmax$. We present the following LP to model our relaxation.

$$\min \alpha \qquad (3)$$

$$u_{i,0} = u_i^{init} \qquad \forall i \in \mathcal{B} \qquad (4)$$

$$\sum_{i=1}^{n} u_{i,l} = u^{end} \qquad (5)$$

$$u_{i,k+1} = u_{i,k} - tr_{i,k} + f_{i,k} \qquad \forall i \in \mathcal{B}, \forall k \in \mathcal{I} \setminus l \qquad (6)$$

$$\sum_{i=1}^{n} tr_{i,k} \leq b_k \qquad \forall k \in \mathcal{I} \qquad (7)$$

$$u_{i,k} \leq C_i \times \alpha \qquad \forall i \in \mathcal{B}, \forall k \in \mathcal{I} \qquad (8)$$

$$\alpha \geq 0, \; u_{i,k} \geq 0, \; tr_{i,k} \geq 0 \qquad i \in \mathcal{B}, k \in \mathcal{I} \qquad (9)$$

Constraints (4) assign the initial memory state to the memory variables at the first breakpoint. We ensure that no bandwidth is lost with constraint (5). The flow conservation is respected with constraints (6) and the bandwidth is shared due to constraints (7). To minimize the highest memory peak, we force the memory state to be lower than or equal to variable α times the buffer capacity (8).

Theorem VII.1. *The optimal value of the Linear Program presented is a lower bound of the peak usage $rmax$.*

Proof. Let $U_i^*(t)$ be the memory usage of buffer i at time t in the optimal solution. Since any solution of oMDPi satisfies the constraints of the linear program, assigning $u_{i,k}$ to $U_i^*(t_k)$ yields a feasible solution of the LP. Moreover, optimal peak usage $rmax^*$ is necessarily greater than or equal to $U_i^*(t_k)$ for any $k \in T$, and hence the optimal value of the LP is lower than or equal to $rmax^*$. ■

VIII. HEURISTIC FOR MULTIPLE WINDOWS

In this section, we present MULTI-WINDOW PRIORITIZE AND INTERRUPT (MWPI), a heuristic method to solve the oMDPi with multiple windows. The main idea of our method is to iteratively use the SINGLEWINDOWSTOP algorithm to compute both a priority and an interruption assignment for each downlink window with respect to a given $rmax$ objective. As our single window algorithm requires handover targets and does not have a long-term view (over multiple windows), we use the LP relaxation presented in Section VII to compute *handover targets* for each downlink window to guide it. As discussed in Section V, those handover targets are a presumed advantageous memory state to reach at the end of each downlink window, for each buffer. Thus, when a handover target is given, the buffers will stop as soon as possible to match this target exactly. However, it is sometimes impossible to reach the handover targets exactly without exceeding $rmax$. In that case, we allow the buffers to exceed their targets, but this can have consequences for the next downlink windows. When no valid solution for a window is found, we backtrack to the previous one and add *handover constraints* to force certain buffers to reduce their memory state. Unlike the handover *targets*, the handover *constraints* cannot be exceeded—i.e. exceeding the handover constraint is considered as an overflow. We keep adding tighter handover constraints until a solution for the current $rmax$ is found or until we backtrack to the first downlink window this way. Let us dive into the algorithm details. We present the pseudo-code of MWPI in Algorithm 3.

We first compute a solution that provides an upper bound on $rmax$ (we re-implemented ITERATEDLEVELING heuristic [6], but any fast method to build a feasible solution can be chosen). As we want to improve this solution, we subtract a small number ϵ from this objective value to set our current $rmax$ target. Our LP provides a lower bound on the highest memory peak and with a handover target for each buffer and each window. We initialize the handover constraint to $h_{i,j}^{ct} = rmax \times C_i, \forall i \in \mathcal{B}, \, j \in \{1, .., m\}$.

In a main loop, we iterate through the downlink windows starting with the first one. At each step, we run SINGLEWINDOWSTOP that builds the solution (priorities and stop) for

Algorithm 3 MWPI: A procedure to find priorities and stops to minimize $rmax$.

Require: An oMDPi instance *ins*, a precision ϵ, a time t^{lim}

1: IteratedLeveling.**Solve**()
2: $rmax \leftarrow$ IteratedLeveling.**GetObjective**()$-\epsilon$
3: $p \leftarrow$IteratedLeveling.**GetPriorities**()
4: $s \leftarrow$**Never**(m, n)
5: LP.**Solve**(*ins*)
6: $rmax^{lb} \leftarrow$LP.**GetObjective**()
7: $h^{tg} \leftarrow$LP.**GetHandover**()
8: **while** $true$ **do**
9: $\quad h^{ct} \leftarrow$**InitHandoverConstraints**$(rmax)$
10: $\quad j \leftarrow 1$, $p^{tmp} \leftarrow$**Equal**(m, n), $s^{tmp} \leftarrow$**Never**(m, n)
11: $\quad$ **while** $true$ **do**
12: $\quad\quad$ **if RunTime**() $\geq t^{lim}$ **then**
13: $\quad\quad\quad$ **return** p, s
14: $\quad\quad$ **if SingleWindow**$(ins_j, h_j^{tg}, h_j^{ct}, rmax)$ **then**
15: $\quad\quad\quad p_j^{tmp}, s_j^{tmp} \leftarrow$**GetSingleWindowSolution**()
16: $\quad\quad\quad$ **if** $j = m$ **then**
17: $\quad\quad\quad\quad p \leftarrow p^{tmp}$, $s \leftarrow s^{tmp}$
18: $\quad\quad\quad\quad rmax \leftarrow$**RunSolution**$(ins, p, s) - \epsilon$
19: $\quad\quad\quad\quad$ **if** $rmax \leq rmax^{lb}$ **then**
20: $\quad\quad\quad\quad\quad$ **return** p, s
21: $\quad\quad\quad\quad$ **break**
22: $\quad\quad\quad$ **else**
23: $\quad\quad\quad\quad j \leftarrow j + 1$
24: $\quad\quad$ **else**
25: $\quad\quad\quad$ **if** $j = 1$ **then**
26: $\quad\quad\quad\quad$ **break**
27: $\quad\quad\quad$ **else**
28: $\quad\quad\quad\quad i \leftarrow$**SelectBuffer**$(1, n)$
29: $\quad\quad\quad\quad val \leftarrow$**SelectValue**$(0, h_{i,j}^{tg})$
30: $\quad\quad\quad\quad j \leftarrow j - 1$
31: $\quad\quad\quad\quad h_{i,j}^{ct} \leftarrow h_{i,j}^{tg} - val$, $h_{i,j}^{tg} \leftarrow h_{i,j}^{ct}$

window j or fails. If the heuristic succeeds in finding a solution with respect to the current $rmax$ target and handover constraints, we advance to the next window. If the last window is reached this way, we simulate the whole instance with the priorities and interruptions to find the $rmax$ value of our solution. Note that the actual $rmax$ can be lower than the $rmax$ target. If $rmax$ matches the lower bound, the current solution is optimal. Otherwise, we set $rmax$ to $rmax - \epsilon$ and restart from the first window. If, at any downlink window j, SingleWindowStop fails, we want to reduce the memory state of at least one overflowing buffer. To do this, we backtrack to the previous downlink window $j - 1$ and randomly select one of the overflowing buffers of window j, and subtract a random value proportional to the current handover target from this buffer handover constraint. We then run SingleWindowStop on window $j - 1$ and repeat the same process. If the first downlink window is reached this way, we cannot backtrack further and so, we perform a restart—we reset the handover constraints and start over at the first downlink window. As our handover constraints can only become tighter, each run is guaranteed to converge.

IX. Experimental Evaluation

We evaluate our heuristic for multiple windows on both Rosetta instances and generated scenarios. Different observation plans at various levels of granularity were computed for the Rosetta mission (see [10] for details). The Long Term Plans (LTP) are refined into Medium Term Plans (MTP), Short Term Plans (STP), and Very Short Term Plans (VSTP), covering four months, one month, a week, and half a week, respectively. In [6], [7], four real scenarios (MTP1–MTP4) were used for evaluation. These instances have 16 buffers and 64–94 downlink windows, with around 10,000 observation events each. As MTP1–MTP3 are solved optimally without interruptions, we focus on MTP4—the only open instance—for comparison with the state of the art. To enhance the evaluation, we generated 90 synthetic scenarios based on Rosetta. The number of buffers ranges from 12 to 20, and the number of downlink windows is 60, 100, or 140. A major drawback in the data set from Rosetta's mission plan is the presence of long non-visibility periods, where high memory peaks are unavoidable, making lower bounds easy to reach with simple strategies (e.g., Downlink Count) without interruptions. To make synthetic instances as challenging as MTP4, we shorten non-visibility periods and proportionally reduce bandwidth during downlink windows. All experiments are run on a single-core Intel E5-2695 v3 2.3 GHz with 32 GB RAM and a 10-minute limit.

TABLE I: Highest memory peak ($rmax$) found for a time limit of 10 minutes on MTP4, for RD [7] and our heuristic MWPI. Best lower bound (LB) on $rmax$ with FullTransfer for RD and our LP relaxation method for MWPI.

Method	Best $rmax$ (%)	Best LB (%)	Time
RD	47.2	46.6	<1 s
MWPI	46.8	46.6	22 s

In Table I, we compare the efficiency of RepairDescent (RD) heuristic [7] and our method to decrease the highest memory peak on MTP4. We also compare a simple lower bound (FullTransfer) used in [7] and the result of our LP relaxation of the problem. The FullTransfer relaxation consists in allocating *all* the bandwidth to every buffer. We show that our method reduces the highest memory peak of the only real instance left open, but at a higher CPU time cost. Our LP relaxation gives the same LB for this instance.

In Table II, we show the average (among our 90 instances) highest memory peak ($rmax$ column), the average lower bound (LB column), the number of proven optimal plans (proven optimal column), the number of proven infeasible plans (proven infeasible column) and the average bandwidth loss for the instances where bandwidth under-use is observed (Band loss column), when using our heuristic against RepairDescent. We can see a clear improvement when using our heuristic to reduce the highest memory peak ($-10.5\,\%$) on our 90 synthetic instances. Our LP relaxation allows us to find better lower bounds and thus to prove optimality more often than Repair Descent. Moreover, we can prove the infeasibility ($rmax > 100\%$) for 13 instances, while Full Transfer relaxation never provides such a proof on our synthetic instances. In our multiple-window heuristic, bandwidth

TABLE II: Average highest memory peak ($rmax$), average best lower bound, number of optimal and proven infeasible solutions, and average bandwidth loss, for MWPI against RD, with a time limit of 10 minutes on 90 synthetic instances.

Method	$rmax$ (%)	LB (%)	Optimal	Infeasible	Band Loss
RD	65.5	42.6	20	0	0%
MWPI	58.6	55.6	37	13	0.02%

loss can occur when a solution is found while some handover targets are exceeded. As interruption variables are not used in REPAIRDESCENT, the loss is 0 for this method. In our instances, bandwidth under-use only occurs for 8 out of our 90 synthetic instances[2]. Among them, the average loss remains very low ($0.2\,\%$) with our method. This corresponds to less than 1.5 hour to dump the residual data for a plan of around a month. If we calculate the average bandwidth loss over all instances, it drops to $0.017\,\%$.

In Figure 3, we plot the average gap to the LP lower bound over time, for the same synthetic instances. We observe that RD is faster to compute acceptable solutions, while our method outperforms it after approximately 10 seconds. This is mainly due to the time spent solving the LP relaxation, which accounts for $55\,\%$ of the resolution time on average. We observe that RD eventually reaches a plateau, likely corresponding to the point where further reduction of $rmax$ becomes difficult using priorities alone. In contrast, our method continues to improve the quality of data transfer plans by leveraging interruption variables. Overall, our method requires more time to generate high-quality plans, but such computation times remain acceptable in offline scheduling contexts—particularly when they yield significant improvements in the objective function.

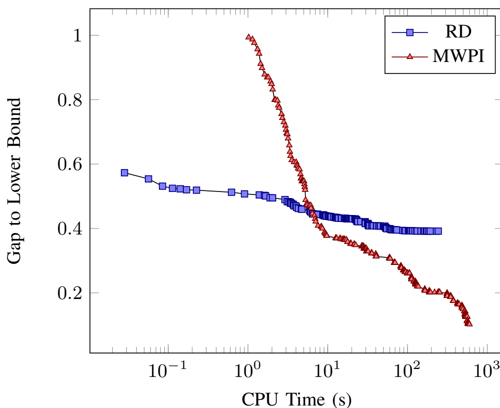

Fig. 3: Average gap to the lower bound over time for RD in blue and MWPI in red. Our method significantly outperforms RD for resolution time of around 10 seconds or more.

[2]We calculate the loss by subtracting the final memory state of REPAIRDESCENT's solution from the final memory state of our solution, the deviation between the two corresponding to the unused bandwidth.

X. CONCLUSION

In this paper, we proposed the first method to solve the overlapping Memory Dumping Problem with interruptions (oMDPi) and demonstrated that using the interruption variables, left aside in all previous works, can improve the transfer plans significantly. More precisely, we have been able to reduce the highest memory peak $rmax$ on the only real instance left open and to improve both the average lower bound ($+30.5\,\%$) and $rmax$ ($-10.5\,\%$) on synthetic scenarios. We also demonstrated that this problem is weakly NP-complete for two or more downlink windows, even with fixed priorities, and we have shown that sometimes, reaching the optimal solution should lead to under-utilization of bandwidth. Thus, depending on future applications, bandwidth loss risk is to be considered carefully when using interruptions, but in our experimental study, the average bandwidth loss remains very low and so the proposed method is clearly appealing for a better data management in future space missions.

The main research direction for oMDPi should focus on complexity analisys for a single downlink window and finding an exact polynomial time algorithm if such a procedure exists. Moreover, using dedicated flow algorithms instead of a linear program for relaxation could improve the resolution speed of our method and seems like a promising lead.

REFERENCES

[1] W. T. Scherer and F. Rotman, "Combinatorial optimization for spacecraft scheduling," in *Proceedings Fourth International Conference on Tools with Artificial Intelligence ICTAI'92*. IEEE, 1992, pp. 120–127.

[2] G. Simonin, C. Artigues, E. Hebrard, and P. Lopez, "Scheduling scientific experiments on the Rosetta/Philae mission," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2012, pp. 23–37.

[3] S. A. Chien, G. Rabideau, D. Q. Tran, M. Troesch, F. Nespoli, M. P. Ayucar, M. C. Sitja, C. Vallat, B. Geiger, F. Vallejo *et al.*, "Activity-based scheduling of science campaigns for the Rosetta orbiter," *Journal of Aerospace Information Systems*, vol. 18, no. 10, pp. 711–727, 2021.

[4] B. Ferrari, J.-F. Cordeau, M. Delorme, M. Iori, and R. Orosei, "Satellite scheduling problems: A survey of applications in Earth and outer space observation," *Computers & Operations Research*, p. 106875, 2024.

[5] A. Oddi and N. Policella, "Improving robustness of spacecraft downlink schedules," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 37, no. 5, pp. 887–896, 2007.

[6] G. Rabideau, S. Chien, M. Galer, F. Nespoli, and M. Costa, "Managing spacecraft memory buffers with concurrent data collection and downlink," *J. of Aerospace Information Systems*, vol. 14, no. 12, pp. 637–651, 2017.

[7] E. Hebrard, C. Artigues, P. Lopez, A. Lusson, S. Chien, A. Maillard, and G. Rabideau, "An efficient approach to data transfer scheduling for long range space exploration," in *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI-22*, L. D. Raedt, Ed. International Joint Conferences on Artificial Intelligence Organization, 7 2022, pp. 4635–4641.

[8] G. Rabideau, S. Chien, F. Nespoli, and M. Costa, "Managing spacecraft memory buffers with overlapping store and dump operations," in *Workshop on Scheduling and Planning Applications, International Conference on Automated Planning and Scheduling (SPARK, ICAPS 2016)*, London, UK, June 2016.

[9] J. Hartmanis, "Computers and intractability: a guide to the theory of np-completeness (michael r. garey and david s. johnson)," *Siam Review*, vol. 24, no. 1, p. 90, 1982.

[10] S. Chien, G. Rabideau, D. Tran, M. Troesch, J. Doubleday, F. Nespoli, M. P. Ayucar, M. C. Sitja, C. Vallat, B. Geiger *et al.*, "Activity-based scheduling of science campaigns for the Rosetta orbiter," in *Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI-15*, Buenos Aires, Argentina, July 2015.