2013 IEEE 54th Annual Symposium on Foundations of Computer Science

An Optimal Randomized Online Algorithm for Reordering Buffer Management

Noa Avigdor-Elgrabli
Computer Science Department
Technion—Israel Institute of Technology
Haifa 32000, Israel
Email: noaelg@cs.technion.ac.il

Yuval Rabani
School of Computer Science and Engineering
The Hebrew University of Jerusalem
Jerusalem 91904, Israel
Email: yrabani@cs.huji.ac.il

***Abstract*—We give an $O(\log\log k)$-competitive randomized online algorithm for reordering buffer management, where k is the buffer size. Our bound matches the lower bound of Adamaszek et al. (STOC 2011). Our algorithm has two stages which are executed online in parallel. The first stage computes deterministically a feasible fractional solution to an LP relaxation for reordering buffer management. The second stage "rounds" using randomness the fractional solution. The first stage is based on the online primal-dual schema, combined with a dual fitting charging scheme. As primal-dual steps and dual fitting steps are interleaved and in some sense conflicting, combining them is challenging. We also note that we apply the primal-dual schema to a relaxation with mixed packing and covering constraints. The first stage produces a fractional LP solution with cost within a factor of $O(\log\log k)$ of the optimal LP cost. The second stage is an online algorithm that converts any LP solution to an integral solution, while increasing the cost by a constant factor. This stage generalizes recent results that gave a similar approximation guarantee using an offline rounding algorithm.**

***Keywords*-online computing, randomized algorithms, reordering buffer management**

I. Introduction

In the reordering buffer management problem (RBM) an input sequence of colored items arrives online, and has to be rescheduled in a permuted output sequence of the same items, with the help of a buffer that can hold k items. The items enter the buffer in their order of arrival. When the buffer is full, one color present in the buffer must be chosen, and the items of this color in the buffer, followed by any new items of the same color encountered along the way, are scheduled in the output sequence one item per time slot, making room for new input items to enter the buffer. The choice of color is made before future input items are revealed. Choosing a color and evicting items is repeated until we reach the end of the input sequence and we empty the buffer. The objective is to minimize the total number of color changes between consecutive items in the output schedule. This seemingly simple model, introduced in [20], formalizes a wide scope of resource management problems in production engineering, logistics, computer systems, network optimization, and information retrieval (see, e.g., [20], [13], [19], [18]). Moreover, beyond its simplicity, elegance, and applicability, the problem turns out to be challenging, and it captures some new and fundamental issues in online computing. In particular, RBM is a very simple model that explores the effect of limited reordering of the input and generalizes lookahead. (We discuss additional unique aspects of RBM below.) We note that the offline version of RBM is NP-hard [6], [15], and there is a polynomial time $O(1)$-approximation algorithm [8].

This paper resolves the randomized competitive ratio of RBM. We design a randomized online RBM algorithm and prove that its competitive ratio is $O(\log\log k)$. This matches the recent lower bound of $\Omega(\log\log k)$ of Adamaszek et al. [2]. All previous online algorithms for RBM are deterministic. A sequence of papers [20], [17], [7], [2] culminated in an $O(\sqrt{\log k})$-competitive algorithm [2], nearly matching the deterministic lower bound of $\Omega\left(\sqrt{\frac{\log k}{\log\log k}}\right)$ in the same paper. Thus, our work is the first to demonstrate an exponential gap between the deterministic and the randomized competitive ratio of RBM. Our algorithm applies the general two-stage approach of the online primal-dual schema (see [14] for a survey). The first stage is a deterministic online algorithm that computes a feasible fractional solution to a time-indexed LP relaxation for reordering buffer management. As we compute the LP solution, we feed it to the second stage—a randomized online algorithm that converts the fractional solution into an integral solution. This algorithm is influenced by our previous paper [8] that gave an offline solution. We employ randomness to replace the prescience needed in the previous result.

The first stage, an online algorithm for computing the LP solution, combines two completely different approaches. One approach is a rather simple charging scheme that penalizes the items in the buffer for each color eviction. In the analysis, the penalties are used to generate a dual feasible solution whose cost is compared to the cost of the algorithm. Applying this approach actually produces an integral solution. The competitive ratio depends on the sizes of the color blocks in the buffer at the times we need to switch colors in the output. If the minimum size block is k', we get a ratio of $O(\log(k/k'))$. Thus, if the color blocks always have at least $\frac{k}{\log k}$ items (at the time of switching colors), this approach alone gives a deterministic $O(\log\log k)$-competitive online algorithm. Of course, there is no way we can guarantee the required condition.

Research supported by Israel Science Foundation grant number 856-11 and by the Israeli Center of Excellence on Algorithms.

0272-5428/13 $26.00 © 2013 IEEE
DOI 10.1109/FOCS.2013.9

The other approach uses a version of the multiplicative weights update method (see [5] for a general survey, and [14] for its application in online computing). In the basic setting of this method, the constraints of a covering LP arrive online, and need to be satisfied as they arrive. In order to satisfy a new constraint, we raise the dual variable that corresponds to the new constraint. This causes dual constraints to get tighter and eventually to become violated. When a constraint becomes tight, the corresponding primal variable is initialized, and then it is increased as an exponential function of the violation of the dual constraint. (The intuition behind using an exponential function comes from viewing the primal variables as the Lagrange multipliers in a lagrangean relaxation of the dual.) At some point, as primal variables grow, the new primal constraint becomes satisfied and the algorithm proceeds to process the next input constraint. The performance guarantees of this method depend on two factors: the ratio between the primal increase and the dual increase, and the scaling factor that is needed to make the dual solution feasible in the end. In essence, as we increase the initialization values of primal variables, the first factor gets worse whereas the second factor improves. This is what governs the performace guarantees that one can prove using this method (e.g. [4], [12], [11], [3]).

Any attempt to adapt the above method to deal with RBM seems to encounter several problems. Unlike other reputed online problems, the irrevocable decisions that an RBM algorithm makes stretch into the future. For example, if a paging algorithm decides to evict a page from the cache, the eviction happens at the time step of the decision. In contrast, if an RBM algorithm decides to evict a color, the eviction may stretch across many time steps starting with the current step, and it might include items that will arrive in the future as the eviction proceeds. In terms of the multiplicative weights update method this means that if it is applied to sensible covering LP relaxations of RBM, then the process at a given time may increase primal variables that correspond to evictions that started in the past, so an online algorithm cannot execute them at the current time. Moreover, the primal variables cannot be initialized in a way that guarantees a ratio better than $\log k$. Even disregarding initialization, the rate of growth of the primal cost cannot be reasonably bounded by the rate of growth of the dual cost. This, too, is the result of having evictions that stretch into future time steps. Roughly speaking, the rate of growth of the dual cost drops as we commit to evict more items in the future, whereas the rate of growth of the primal cost depends only on what's been evicted until the current step.

Very recently, Adamaszek et al. [3] implemented the multiplicative weights update method to solve a problem they call *buffer scheduling for block devices*. This is a variant of RBM that differs from it in one crucial aspect. In block devices, while evicting a color from the buffer, new arriving items of that color cannot be appended to the output sequence without incurring additional cost. For example, if the entire input sequence consists of a single color, an RBM solution pays 1, while a block device solution pays approximately $\frac{n}{k}$. The Adamaszek et al. result, an $O(\log \log k)$-competitive randomized online algorithm for buffer scheduling for block devices, partially overcomes the above problems (but not for RBM). They do this by employing a resource augmentation argument, adapted from [17], [16]. The original argument in the referenced papers shows that (for RBM) replacing a size k buffer by a size $k' = \frac{k}{4}$ buffer increases the optimal cost by a factor of $O(\log k)$. This is known to be tight for $k' = \frac{k}{4}$ [1]. Adamaszek et al. adapt the proof to block devices and generalize it to show that replacing a size k buffer by a size k' buffer, for $k' = \left(1 - \frac{O(1)}{\ln k}\right) \cdot k$, increases the optimal cost by a constant factor.

Intuitively, resource augmentation equips the online algorithm with some lookahead of $k - k'$ items, and this lookahead allows the algorithm to decide on up to $k - k'$ steps into the future. Technically, the gap enables an initialization of the primal variables to roughly $\frac{k-k'}{k}$ instead of $\frac{1}{k}$, and this reduces the competitive ratio that one can hope for from $\log k$ to $\log(k/(k-k'))$. (Setting the multiplicative weights update method to prove this claim is the main contribution of [3].) Moreover, the gap helps bound the rate of growth of the primal cost beyond the cost of initializations. However, this argument only works if none of the (fractionally) evicted color blocks have more than $O(k - k')$ items. Clearly, this assumption may fail. In the case of block devices, Adamaszek et al. overcome this problem by generating an infeasible primal solution. Thanks to the fact that in the block devices setting, when a color is evicted, even though its schedule stretches into the future, it does not include any future items, they are still able to round the infeasible solution to get a feasible integral solution.

Unfortunately, this approach does not work in the case of RBM. Instead, we combine two methods, as mentioned above. One method is a multiplicative weights implementation that uses the above resource augmentation idea and works for color blocks with at most $k - k'$ items in the buffer. Another method is the dual fitting approach that works for color blocks with at least $k - k'$ items in the buffer. Combining the two methods is challenging, and is the main technical contribution of this paper. One reason why the combination is non-trivial is that at any given time the buffer can be in a "mixed" state with both small and large color blocks, but each method works on the entire buffer. Also, the dual fitting approach inherently generates an integral solution, whereas the multiplicative weights approach inherently generates a fractional solution. Thus, we have to decide if a color block will turn out to be small or large before we know how many items of this color we'll accumulate in the buffer by the time we complete its eviction. If we start removing it fractionally using multiplicative weights, we cannot regret this decision later and switch to dual fitting. Another difficulty is that the dual fitting approach forces us to use a mixed packing-covering relaxation for RBM, where in each time step we are presented with a pair of primal constraints—a packing constraint and a covering constraint. Essentially all past work uses packing or covering linear programs, or small variations thereof (such

as having additional box constraints).[1] To the best of our knowledge, prior to our work no relaxation of similar form to ours was known to be amenable to the online primal-dual schema.

The rest of the paper is organized as follows. After introducing some notation, definitions, and an overview in Section II, we present our online multiplicative weights algorithm in Section III, and our online rounding algorithm in Section IV. Due to page limits, some of the proofs are omitted. They can be found in the full version of the paper posted on Arxiv [9]. Part of the multiplicative weights analysis is given in Section V. The rounding analysis is omitted. The explicit constants in the rest of the paper are somewhat arbitrary. We made no attempt to optimize them.

II. Preliminaries

Let $\mathcal{I}$ be a sequence of colored items. We denote the color of an item i by $c(i)$. Abusing notation, we denote the color of a sequence I of items of the same color by $c(I)$. We denote by $\text{OPT}_k(\mathcal{I})$ the cost of an optimal (offline) RBM schedule of $\mathcal{I}$ using a buffer of size k. The following lemma is adapted from [16], [3].

Lemma 1. *For every input sequence $\mathcal{I}$ and for every $k' < k$, $\text{OPT}_{k'}(\mathcal{I}) \leq \frac{2k+(k-k')\ln k'}{k'} \cdot \text{OPT}_k(\mathcal{I})$.*

In our algorithm and analysis we use this lemma with $k' = k - \frac{2k}{\ln k}$, which increases the optimal cost by a constant factor.

Consider a sequence I of items of a single color c in $\mathcal{I}$ that includes all the items of this color between the first and last item of I. If there is an RBM solution that outputs I starting at time j, we call the pair (I, j) a *batch*. Thus, an RBM solution consists of scheduling or packing batches in the interval of output time slots $\{k+1, k+2, \ldots, k+n\}$, where every output slot is used by at most one batch, and every input item is scheduled or covered by at least one batch. In other words, an RBM solution is a bipartite matching of input items to output slots. The matching must observe the order of input on each color separately, and an item cannot be matched to an output slot that precedes its arrival. The cost is the number of batches, where a batch is a maximal output interval that got matched to a set of items of the same color. This discussion leads us to a natural linear programming relaxation for RBM. We can think of the output slots as a channel of width 1 spanning the output interval $\{k+1, k+2, \ldots, k+n\}$. We pack batches fractionally in this channel, without violating the width constraint, but covering all input items at least once. This relaxation is essentially identical to the one used in [7], [8].

[1]The one very recent exception [10] indeed deals with mixed packing-covering LPs. However, the objective of that paper is to minimize the violation of the packing constraints, and therefore seems irrelevant to RBM.

We denote it simply by LP_k. Formally, LP_k is

$$\text{minimize} \sum_{(I,j)} x_{I,j} \text{ subject to}$$

$$\sum_{(I,j):\ i \in I} x_{I,j} \geq 1 \qquad \forall i = 1, 2, \ldots, n \qquad (1)$$

$$\sum_{(I,j'):j' \leq j < j'+|I|} x_{I,j'} \leq 1 \quad \forall j = k+1, \ldots, k+n \quad (2)$$

$$x \geq 0.$$

Here, $x_{I,j}$ is the weight of the batch (I, j) in the packing. Constraints (1) require that every item is eventually removed from the buffer (in batches of total weight 1). Constraints (2) restrict the output to remove a total weight of at most 1 in each time slot. The dual linear program, which we denote by DP_k is

$$\text{maximize} \sum_{i=1}^{n} y_i - \sum_{j=k+1}^{k+n} z_j \text{ subject to}$$

$$\sum_{i \in I} y_i - \sum_{j'=j}^{j+|I|-1} z_{j'} \leq 1 \quad \forall (I, j) \qquad (3)$$

$$y, z \geq 0.$$

Our algorithm computes online an LP_k feasible solution x and a $\text{DP}_{k'}$ feasible solution (y, z). The algorithm feeds x, as it is being produced, to an online "rounding" procedure that produces an LP_k feasible integer solution $\bar{x}$, which is the output of our online algorithm. Our main result, which the rest of the paper builds towards, is

Theorem 2. *There is an $O(\log \log k)$-competitive randomized online algorithm for RBM.*

Proof: Theorem 3 establishes that the value of x is at most $O(\log \log k)$ times the value of (y, z), which is a lower bound on $\text{OPT}_{k'}$, and hence at most $O(\text{OPT}_k)$ (by Lemma 1). Lemma 5 establishes that the value of $\bar{x}$ is at most $O(1)$ times the value of x, and this concludes the proof of the theorem. ∎

III. Solving the LP Online

In this section we give an online algorithm that constructs a primal feasible solution x to LP_k and a dual feasible solution (y, z) to $\text{DP}_{k'}$. In Section V we prove the following theorem.

Theorem 3.

$$\sum_{(I,j)} x_{I,j} \leq O(\log \log k) \cdot \left(\sum_{i=1}^{n} y_i - \sum_{j=k'+1}^{k'+n} z_j \right).$$

We construct simultaneously a feasible primal solution x, an infeasible dual solution $(\hat{y}, \hat{z})$, and an auxiliary dual penalty $\bar{y}$. The construction of $(\hat{y}, \hat{z})$ uses a non-trivial implementation of the multiplicative weights update method, and $\bar{y}$ is generated by a dual fitting argument. A feasible dual solution (y, z) can be derived by scaling down $(\hat{y} + \bar{y}, \hat{z})$ by a factor of $O(\log \log k)$. The algorithm maintains throughout its execution for every color c an index s_c which is the earliest

item of color c whose primal constraint (1) is violated, i.e., $\sum_{(I,j):\ s_c \in I} x_{I,j} < 1$.

Notice that if a color c is not present in the buffer (for instance, c has not been encountered yet), the algorithm may not know s_c. However, if the buffer has no item of color c, then the algorithm does not use s_c, so this does not cause a problem. The algorithm further maintains the earliest output slot t whose primal constraint (2) is not tight, i.e., $\sum_{(I,j):\ j \le t < j+|I|} x_{I,j} < 1$. Initially, t is set to $k+1$.

The dual solution $(\hat{y}, \hat{z})$ is generated as follows. Initially, all dual variables are set to 0. The solution is parametrized by μ, which is raised at a uniform rate. We occasionally refer to μ as *time*, but this should not be confused with the discrete input and output time steps. Further notice that even though for convenience we describe the algorithm as a continuous process, it can be discretized easily, and it can be implemented efficiently (regardless, competitive analysis is not concerned with computational efficiency). The algorithm raises all the variables $\hat{y}_i$ with $i \ge s_{c(i)}$ and all the variables $\hat{z}_j$ for $j \ge t$ at the same rate $d\mu$. (This raises also future $\hat{y}_i$-s and $\hat{z}_j$-s; when we reach them, we will initialize their value to what's determined by this process.) Notice that we raise the $\hat{y}_i$-s corresponding to violated primal constraints (1), and the $\hat{z}_j$-s corresponding to primal constraints (2) that are not tight. Raising dual variables causes x to change, thus removing items fractionally or integrally from the algorithm's buffer. This eventually increments the s_c-s and t, thus changing the set of dual variables that are raised. It also affects $\bar{y}$. The process ends when t passes past time $k'+n$. At this point, we simply evict the remaining buffer contents, using the output slots up to time $k+n$. (Notice that this last step does not cost more than the total number of colors plus one; the total number of colors is a lower bound on the optimal cost.)

We now explain how raising $(\hat{y}, \hat{z})$ affects x. At any given time, let B denote the set of items encountered so far, whose primal constraints are violated, and let B_c denote the set of items of color c in B. In other words, B_c includes all the color c items that appear in the input sequence starting from s_c and before the current slot t. Notice that for every item in B, at least a fraction of that item is still in the algorithm's buffer. There may be additional items in the algorithm's buffer. These are items that are already scheduled to be removed entirely from the buffer, but t hasn't yet passed the point where they disappear from the buffer. The items in B are endowed with one of three states: *fractional*, *integral*, or *frozen*. If any item in B_c is integral, then they all are. Otherwise, the first ones are fractional and the remaining ones (if any) are frozen. We will refer to a set of items in B of the same color and the same state as a *block*. Thus, B_c consists of either one or two blocks: an *active* block B_c^{act} that is either fractional or integral, and a frozen block B_c^{frz} that might be empty (and must be empty if B_c^{act} is integral). With a slight abuse of terminology, we sometimes also refer to all of B_c as a block. These sets (B, B_c, B_c^{act}, B_c^{frz}) are all functions of μ (and so are s_c, t, x, $\hat{y}$, $\hat{z}$, $\bar{y}$, and other variables defined below).

Consider a dual constraint indexed (I,j) and put $c = c(I)$. Let

$$\sigma_{I,j} = \sum_{i \in I} \hat{y}_i - \sum_{j'=j}^{j+|I|-1} \hat{z}_{j'}$$

denote the current *dual cost* of the batch (I,j). Notice that we know this value at any time μ, even if the batch is matched to output slots we haven't yet reached.

Fact 4. *Consider a batch (I,j) of color c. If $\frac{d\sigma_{I,j}}{d\mu} > 0$ then there must be an item $i \in B_c \cap I$ that is matched by (I,j) to an output slot before the current time t.*

Proof: If all the items in $B_c \cap I$ are matched by (I,j) at time t or later, then consider $i \in B_c \cap I$ which is matched by (I,j) to $j' \ge t$. Both $\hat{y}_i$ and $\hat{z}_{j'}$ increase at the same rate. Therefore, $\sigma_{I,j}$ cannot increase. ∎

The algorithm produces a primal solution x by scheduling batches of items, i.e., by raising $x_{J,t}$, for some batches (J,t), where $t = t(\mu)$. If we schedule a batch (J,t) of color c, then J begins with the items in B_c^{act} (at the time μ when (J,t) is scheduled) and J could extend beyond B_c^{act}. We extend J online by appending new items of color c as they arrive according to the rules below. (Technically, appending a new item means that we move the current weight of $x_{J,t}$ to $x_{J',t}$, where J' is the new set with the new appended item.) We append a new item i of color c to J if and when the following becomes true: i's state is the same as the state of the previous items in J (when they were added to J), and we did not pass beyond the end of the current schedule of J. In particular, when an item extends J it is in B_c^{act}. Specifically, we do not append i to J, even though it is in our buffer by the time we reach the end of the current schedule of J, if $i \in B_c^{frz}$ at that time. To summarize, scheduling a batch of color c at time μ involves packing in the output stream, starting with output slot $t = t(\mu)$, the sequence of items in $B_c^{act} = B_c^{act}(\mu)$ (with a weight that cannot be greater than the remaining unscheduled weight of the first item in B_c^{act}), and later possibly extending this sequence with new items on-the-fly.

The regular execution of the algorithm is to schedule continuously batches for every color c for which B_c^{act} is fractional. The rate $dx_{J,t}$ at which we raise $x_{J,t}$ is governed by pseudo-dual cost variables $\hat{\sigma}_{I,j}$ and pseudo-primal variables $\hat{x}_{I,j}$, defined for all batches (I,j). We maintain the equation

$$\hat{x}_{I,j} = \begin{cases} \frac{1}{\ln k} \cdot \hat{\sigma}_{I,j} & \hat{\sigma}_{I,j} < 1, \\ \frac{1}{\ln k} \cdot e^{\hat{\sigma}_{I,j}-1} & \hat{\sigma}_{I,j} \ge 1. \end{cases}$$

We set

$$\frac{dx_{J,t}}{d\mu} = \max_{(I,j)} \left\{ \frac{d\hat{x}_{I,j}}{d\mu} :\ c(I) = c(J) \right\}.$$

Notice we schedule batches simultaneously for all colors with fractional items in the buffer.

In order to complete the description of the algorithm's regular execution, we need to explain how $\hat{\sigma}_{I,j}$ changes. Initially, $\hat{\sigma}_{I,j}$ is set to 0, and at certain events (see below) we reset $\hat{\sigma}_{I,j}$ to 0. (Notice that this resets $\hat{x}_{I,j}$ to 0 retroactively, but the reset has no effect on the current partial solution x that we have already constructed, only on future updates, so x is indeed constructed online.) During an interval $[\mu_1, \mu_2]$ with no

reset, $\hat{\sigma}_{I,j}$ does not decrease. In order to explain the increase in $\hat{\sigma}_{I,j}$, consider the increase in $\sigma_{I,j}$ when μ changes by an infinitesimal amount $d\mu$. Let $t = t(\mu)$, and let $c = c(I)$. If $t \geq j$ or if all the items in $B_c \cap I$ are matched by (I,j) at time t or later, then $\sigma_{I,j}$ does not increase, and $\hat{\sigma}_{I,j}$ does not change. Otherwise, $\sigma_{I,j}$ increases by $d\mu$ times the number of items in $B_c \cap I$ that are matched by (I,j) before time t. In this case, $\hat{\sigma}_{I,j}$ increases by $d\mu$ times the number of items in $B_c^{act} \cap I$ that are matched by (I,j) before time t. We will later see that in this case $d\hat{\sigma}_{I,j} \geq \frac{10}{11} \cdot d\sigma_{I,j}$. We say that a batch (J,t) of color c that is scheduled during this increase is *relevant to* (the dual cost of the batch) (I,j). Intuitively, if (J,t) is relevant to (I,j), then when $x_{J,t}$ increases, $\hat{x}_{I,j}$ increases by at most the same amount. Notice that if a batch (J,t) that is relevant to (I,j) is scheduled without interruption (i.e., it never reaches an item that is frozen at the time slot it needs to be scheduled), then J includes the last item of I.

Occasionally during regular execusion, we reset $\hat{\sigma}_{I,j}$ to 0. We call this a *regular reset* (to distinguish it from other resets that happen when regular execusion is interrupted). This happens in the following situation. Let the current time be μ. Let $f = f(I,j) \in I$ be the first item that *interrupts* (is not appended to) a scheduled batch (J,t') $(t' < t(\mu))$ that is relevant to (I,j), because $f \in B_c^{frz}$ when it needs to be appended. If at time μ the number of items in B_c that arrived before f just dropped below $\frac{1}{2}|B_c|$, we reset $\hat{\sigma}_{I,j}$ to 0. Notice that we do this only for the first such item $f \in I$, so for any batch (I,j), we do a regular reset at most once. We denote the time of the regular reset by $\mu_0(I,j)$. If (I,j) never experiences a regular reset, we put $\mu_0(I,j) = \infty$. Also notice that if $\hat{\sigma}_{I,j}$ is reset to 0, automatically $\hat{x}_{I,j}$ is reset to 0.

Regular execution is interrupted in a few cases as follows. Upon interruption, we keep executing the valid cases until none of them hold, in which case regular execution is resumed.

Case 1: A primal constraint (1) becomes satisfied, i.e., for some $i \in B$, $\sum_{(I,j):\ i \in I} x_{I,j}$ reaches 1. In this case we increment $s_{c(i)}$. Notice that this also changes B_c^{act}.

Case 2: A primal constraint (2) becomes tight, i.e., $\sum\{x_{I,j} : j \leq t < j + |I|\}$ reaches 1. In this case, we increment t. Each new item that enters the buffer initializes its state as follows. If there are integral items of the same color, it enters the buffer as integral. Otherwise, it enters the buffer as frozen (however, the frozen state may change immediately due to the application of one of the following cases).

Case 3: If $|B_c^{frz}| > \frac{k}{100 \ln k}$ for some color c, we schedule all the remaining volume of B_c^{act}. (This may involve scheduling several distinct batches, and it also increments s_c.) Then, we change the state of all the items in B_c^{frz} to integral. (In particular this moves all of them to B_c^{act}.) Finally, we reset $\hat{\sigma}_{I,j}$ (and hence $\hat{x}_{I,j}$) to 0 for all batches (I,j) of color c.

Case 4: If B_c^{act} is fractional and $|B_c^{act}| < \frac{k}{10 \ln k}$, we change the state of all the items in B_c^{frz} to fractional (in particular, they move to B_c^{act}).

Case 5: There is an integral block B_c^{act}, and $\hat{\sigma}_{I,j}$ reaches 1 for a color c batch (I,j). (Notice that taking into account the reset in Case 3 above, we can assume that I is a subset of B_c^{act}.) We *suspend* all fractional scheduled batches that haven't yet ended. We set $\bar{y}_i = \frac{1}{2|B_c^{act}|}$ for every $i \in B_c^{act}$. We schedule, starting at the current t, an integral (weight-1) batch with all the items in B_c^{act} followed by any items that can be appended to that block while it is being evicted from the buffer. We reschedule the unfinished portion of the suspended batches following this integral batch. Finally, we reset $\hat{\sigma}_{I,j}$ (and hence $\hat{x}_{I,j}$) to 0 for all batches (I,j) of color c. Notice that following this case, both s_c and t are incremented by at least $|B_c^{act}|$. See Figure 1.

Case 6: Since the last application of this case, we've moved past the end of regular execution fractionally scheduled batches of color c with total weight at least $\frac{1}{10}$ (suspended batches are not considered to have ended). We apply the same procedure as in Case 5 to the block B_c^{frz}, except that we don't raise $\bar{y}$. To distinguish batches scheduled by this case from integral batches scheduled by Case 5, we will refer to the ones we schedule here as *weight*-1 *fractional batches*.

IV. Online Rounding

In this section we give a randomized online algorithm that rounds the fractional solution x to an integral solution for the reordering buffer management problem. In the full version [9] we prove the following Lemma.

Lemma 5. *The expected cost of the solution generated by the online rounding algorithm is $O(1) \cdot \sum_{I,j} x_{I,j}$, where x is the input to the algorithm (a feasible solution of LP_k).*

The rounding algorithm presented here is inspired by our deterministic offline rounding algorithm in [8]. Here we use randomness to replace the knowledge of future input that is needed in [8]. Whenever our rounding algorithm needs to choose a color to evict, say at output slot t, the algorithm uses only the input up to time t and the fractional solution x that we computed up to time t. Thus, our randomized online algorithm for reordering buffer management repeats two alternating steps: (i) Extend the fractional solution deterministically up to the current time. (ii) Evict from the buffer using randomness some items chosen based on the current partial fractional solution. This increments the current time to the next vacant output slot.

The rounding algorithm works in phases. The first phase begins at time $k+1$. In the beginning of a phase, the algorithm chooses one or more color blocks to evict, based on the fractional solution x that was computed up to the output time slot t where the phase begins. Then, the algorithm evicts the chosen blocks, and a new phase begins. Notice that in order to execute the next phase, we need to extend the fractional solution x to the new time slot that we have reached, taking into account the new input items that have entered the buffer during the last phase.

In choosing the colors to evict in a phase, we consider four cases. Let $\delta > 0$ be a sufficiently small constant, and let t_0 be the starting output time slot of the current phase. More precisely, the fractional solution computed so far fully uses the time slots up to at least t_0, whereas the integral solution computed so far extends up to time slot $t_0 - 1$.

Case 1: The buffer contains an item from which the fractional solution removed so far a weight of at least δ. We evict the color block of this item.

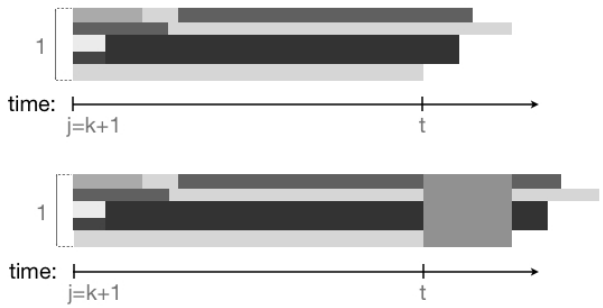

Fig. 1. Scheduling an integral batch at time slot t (case 5 and case 6). Notice that the suspended fractional batches are continued after the integral block. (The strips of varying tones and sizes indicate the scheduled batches.)

Case 2: The total weight of the items that the fractional solution schedules in the time slot t_0 and are also in our buffer is at least 2δ. We choose one such item at random with probability proportional to the weight it is removed at time t_0, and we evict its block.

Case 3: A weight of more than $\frac{1}{2}$ of the items that the fractional solution schedules at time t_0 belong to a single color c that we just evicted from our buffer (i.e., the integral solution evicts at time slot $t_0 - 1$ an item of color c). In this case we first choose color blocks to evict according to the following procedure, and then we evict all these blocks in arbitrary order.

We now describe the procedure for choosing color blocks to evict in Case 3. Besides choosing color blocks, the procedure also "locks" some volume fractionally scheduled before time t_0. Any volume that is fractionally scheduled starts unlocked. Locked volume is assigned to a specific evicted block, and when the weight in the fractional buffer of an item in this block drops below $1 - \delta$, the volume assigned to this block becomes unlocked again.

We partition the colors into classes according to the number of items in the buffer of each color at time t_0. A color c is in class $s = 1, 2, \ldots, \log k + 1$ iff the number of items in the buffer of color c is in $[2^{s-1}, 2^s)$. Next, we partition the classes into subclasses as follows. For every color c let w_c denote the average over the color c items in the buffer of the unlocked volume that the fractional solution scheduled for this item before time t_0. Let W_s denote the sum of w_c over all colors c in class s. To construct a subclass, we collect blocks until their total w_c weight exceeds δ. In a class, we construct disjoint subclasses using this process while the remaining weight is at least δ. Notice that because $w_c < \delta$ for every color c, the total weight of a subclass is in $[\delta, 2\delta)$. Also notice that in each class we might have colors with total w_c weight of less than δ that are not assigned to subclasses. We ignore those colors. If $W_s < \delta$ then no block of class s is chosen. In each subclass, we choose at random one color block to evict. The probability of choosing a color c is proportional to w_c. The chosen block locks all the unlocked volume in the subclass. Finally, we also choose the largest color block in our buffer (This block takes care of the excess weight that we ignored in the above choice.)

Case 4: If all else fails (i.e., for all previous cases, the conditions for executing the case do not hold), we choose the largest and second largest color blocks, and also apply the Case 3 procedure that chooses more colors. If after evicting the largest or second largest color block one of the other cases applies, we terminate the phase without evicting the remaining chosen blocks. (If we don't get to evict the Case 3 procedure choices, we annul the locks generated by the choice.) We stress that we choose all the blocks to evict in this case according to the situation at time t_0, but some of the chosen blocks might end up not being evicted.

V. Analyzing the LP Algorithm

We first observe that by the definition of the algorithm (Case 1 and Case 2) it constructs a feasible fractional solution.

Observation 6. *The primal solution x is a feasible solution of* LP_k.

We now bound the total volume of items in the buffer that the algorithm schedules at any given time while in regular execution.

Claim 7. *If B_c^{act} is fractional, then $|B_c| < \frac{12k}{100 \ln k}$ and $|B_c^{act}| < \frac{11k}{100 \ln k}$.*

Proof: By Case 3, if B_c^{act} is fractional, then $|B_c^{frz}| \leq \frac{k}{100 \ln k}$. By Case 4, we keep new items of color c in B_c^{frz}, unless $|B_c^{act}| < \frac{k}{10 \ln k}$. If B_c^{act} drops below $\frac{k}{10 \ln k}$, we move the items in B_c^{frz} to B_c^{act}, adding at most $\frac{k}{100 \ln k}$ new items, so $|B_c^{act}| < \frac{11k}{100 \ln k}$ always (while fractional). Combining this bound with the bound on B_c^{frz}, we get that $|B_c| < \frac{12k}{100 \ln k}$. ∎

Corollary 8. *At any time μ, the volume of items in $B(\mu)$ that is scheduled beyond time $t(\mu) - 1$ is less than $\frac{12k}{100 \ln k}$.*

Proof: There is a total weight of less than 1 of scheduled batches that extend to time $t = t(\mu)$ and beyond (otherwise, Case 2 would have happened). Each of these batches is fractional, so by Claim 7 it has less than $\frac{12k}{100 \ln k}$ items that are in $B(\mu)$. Therefore, the total volume that is scheduled of items in $B(\mu)$ is less than $\frac{12k}{100 \ln k}$. ∎

Claim 9. *Consider the (partial) solution x at a time of regular execution of the algorithm. The total volume that the algorithm already scheduled of items that are currently in B is bounded by*

$$\sum_{(I,j)} (x_{I,j} \cdot |B \cap I|) < |B| - k'.$$

Proof: Let μ denote the current time, and let $t = t(\mu)$ denote the current output time slot. By Corollary 8, the total volume of items in B that is scheduled beyond time $t-1$ is less than $\frac{12k}{100 \ln k}$. The total volume that is scheduled before time t is exactly $t-1-k$. By the definition of B, exactly $t-1-|B|$ items are scheduled to be removed completely from the buffer. Therefore, the volume that is scheduled from items in B is bounded as follows:

$$\sum_{(I,j)} (x_{I,j} \cdot |B \cap I|) < (t-1-k) - (t-1-|B|) + \frac{12k}{100 \ln k} =$$

$$= |B| - k + \frac{12k}{100 \ln k} = |B| - \left(k' + \frac{2k}{\ln k}\right) + \frac{12k}{100 \ln k} < |B| - k'.$$

■

Claim 10. *For every batch (I,j), it holds that $\hat{x}_{I,j} \leq \frac{11}{10}$ always.*

Proof: Let $c = c(I)$. Notice that $\hat{x}_{I,j}$ can increase beyond 1 only while B_c^{act} is fractional. Consider an interval $[\mu_1, \mu_2]$ of uninterrupted regular execution where $\sigma_{I,j}$ increases by δ and the output time slot is t. Let (J,t) be a scheduled batch relevant to (I,j). Notice that $x_{J,t}$ is at least the total increase in $\hat{x}_{I,j}$ during the interval $[\mu_1, \mu_2]$ where $x_{J,t}$ was set. We prove the claim by bounding the total increase in $x_{J,t}$ for all relevant (J,t).

First notice that the total increase for all $x_{J,t}$ that extend all the way to the last item in I must be at most 1. This is because after an increase of 1 the last item of I is no longer in B, and $\sigma_{I,j}$ cannot increase further. (A batch that is suspended and later rescheduled is considered here as a single batch.) The only reason that we do not extend J to the last item of I is if some item along the way is frozen at the time it needs to be appended to J. If when we reach the end of J's schedule, we've accumulated a cost of at least $\frac{1}{10}$ of interrupted schedules since the last time a weight-1 fractional batch of color c was scheduled, then we schedule a weight-1 fractional batch beginning with B_c^{frz}. Notice that the items of B_c^{frz} are scheduled past where they are matched by (I,j), so all the remaining items of I will be evicted from the buffer completely, and $\sigma_{I,j}$ will not increase further.

So consider the first interrupted such (J,t) (so the first color c item following J is $f(I,j)$). At the time μ when the interruption occurs, all the items in $B_c^{frz}(\mu)$ are not appended to any scheduled batch. Let μ' denote the time when the items that were in $B_c^{frz}(\mu)$ are schduled to be removed entirely from the buffer (i.e., they are removed from B). Every scheduled batch that removes these items must schedule them after time slot $t(\mu)$, so unless such a batch is interrupted, it includes the last item of I.

Suppose that between time μ and time μ' no other scheduled batch of color c is interrupted. If $t(\mu')$ is past the end of the first scheduled batch that removes $B_c^{frz}(\mu)$, then this batch is not interrupted and thus it includes the last item of I. Therefore, none of the scheduled batches that remove $B_c^{frz}(\mu)$ are interrupted before they include the last item of I. Otherwise, at time μ' there must be a total weight of 1 of scheduled batches of this color, because each item in $B_c^{frz}(\mu)$ is scheduled with total weight 1 in batches that begin past $t(\mu)$, and none of these batches are interrupted until $t(\mu')$ (by our above assumption). In this case, at time μ' all of $B_c^{act}(\mu')$ is scheduled to be removed completely from the buffer. Also, it must be that $B_c^{frz}(\mu') = \emptyset$, because if B_c^{frz} was non-empty just before μ', it is moved to B_c^{act} due to Case 4 of the algorithm (as $|B_c^{act}|$ drops to 0). Therefore, while all these batches are still being scheduled, any new item of color c is appended to all of them and is thus removed from the buffer, so none of them are interrupted at least until the first one ends. As the first such batch that ends must include the last item of I, they all must include the last item of I.

If there is an interruption between time μ and time μ', repeat this argument for the new interrupted batch (J,t) and interrupting B_c^{frz}. Notice that any such interruption must have the property that (J,t) must schedule the previous B_c^{frz} past $t(\mu)$, and therefore past where it is matched by (I,j). If we accumulate $\sum_{J,t} x_{J,t} \geq \frac{1}{10}$ of interrupted (J,t) at some point, then we schedule a weight-1 fractional batch, and by the argument above $\sigma_{I,j}$ does not increase further. Notice that in this case the last such $x_{J,t} \leq 1$ so $\hat{x}_{I,j} < \frac{11}{10}$. Otherwise, the weight of interrupted (J,t) is less than $\frac{1}{10}$ and the weight of uninterrupted (J,t) is at most 1, so $\hat{x}_{I,j} < \frac{11}{10}$. (Notice that along the way we might have reset $\hat{x}_{I,j}$, but this can only decrease its value, and we analyzed aggregate increase $\hat{x}_{I,j}$.) ■

A. Dual feasibility

The main technical difficulty is to show that the dual solution that the algorithm computes is a feasible solution. In order to prove this, we need to show that the constraints (3) are satisfied, namely that for every batch (I,j),

$$\sum_{i \in I} y_i - \sum_{j'=j}^{j+|I|-1} z_{j'} \leq 1.$$

We consider several cases in the following claims. These cases will be combined in the pursuing proof of Lemma 15. Consider a batch (I,j). The items in I are partitioned by the algorithm's execution into segments. A segment is a maximal substring of items with the same state when removed from the buffer. Thus, there are alternating fractional and integral segments. An integral segment consists of a block of items that were removed together in a single application of Case 5 of the algorithm. In between two integral segments (or an integral segment and an endpoint of I, or two endpoints of I) there is a fractional segment.

We first deal with batches that do not contain an integral segment.

Claim 11. *For every batch (I,j) for which all of I is one fractional segment,*

$$\sigma_{I,j} = \sum_{i\in I} \hat{y}_i - \sum_{j'=j}^{j+|I|-1} \hat{z}_{j'} = O(\log\log k).$$

Proof: Denote $c = c(I)$. Notice that $\sigma_{I,j}$ increases only when $t > j$ and $s_c \in I$ (this is a necessary but not sufficient condition). We bound the total increase in $\sigma_{I,j}$, ignoring possible decreases along the way. Therefore, we may assume that I is a maximal set without an integral segment, because extending it backwards and forwards can only make the sum of increases larger. To see this, notice that if $t > j$, then extending I backwards adds items whose $\hat{y}$ value possibly increases, whereas its corresponding $\hat{z}$ value remains fixed. Extending I forwards adds items whose $\hat{y}$ value definitely increases (because $s_c \in I$), and its corresponding $\hat{z}$ value possibly also increases.

Notice that there is at most one value $\mu_0 = \mu_0(I,j)$ of μ where $\hat{\sigma}_{I,j}$ (and therefore $\hat{x}_{I,j}$) is reset to 0 while $s_c \in I$, because I does not contain an integral segment. Recall that $f = f(I,j)$ is the first item in I that is in B_c^{frz} when we need to append it to a relevant scheduled batch. Then μ_0 is the smallest value of μ for which $|\{i \in B_c : \ i \geq f\}| \geq \frac{1}{2}|B_c|$.

Notice that whenever $\sigma_{I,j}$ increases by δ, then $\hat{\sigma}_{I,j}$ increases by at least $\frac{10}{11}\cdot\delta$. This is because the increase in $\hat{\sigma}_{I,j}$ is incurred by all the items that increase $\sigma_{I,j}$, excluding those that are currently frozen. However, $|B_c^{act}| \geq \frac{10}{11}\cdot|B_c|$ and if (I,j) matches any item of B_c^{frz} before the current time t, then it matches all the items of B_c^{act} before time t as well. (As B_c^{act} is fractional and (I,j) is maximal, $B_c^{act} \subseteq I$.)

Notice that

$$\hat{\sigma}_{I,j} = \begin{cases} \hat{x}_{I,j}\cdot\ln k & \hat{x}_{I,j} \leq \frac{1}{\ln k}, \\ 1 + \ln\hat{x}_{I,j} + \ln\ln k & \text{otherwise.} \end{cases}$$

By Claim 10, $\hat{x}_{I,j} \leq \frac{11}{10}$ always. Therefore, $\hat{\sigma}_{I,j} < 2 + \ln\ln k$ always. Because $\hat{\sigma}_{I,j}$ is reset at most once, we get that $\sigma_{I,j} \leq 2\cdot\frac{11}{10}\cdot\max\{\hat{\sigma}_{I,j}\} < \frac{22}{5} + \frac{11}{5}\cdot\ln\ln k$. ∎

The proof of the following property is useful in the rest of the analysis. Consider a batch (I,j) of color c. Let $I_1, I_2, \ldots I_m$ be its integral segments (by the order of the matching). Let j_r be the time that the first item of I_r is matched by (I,j), and let t_r be the time slot where I_r was scheduled by the algorithm. Denote by $\Delta_r = j_r - t_r$ the difference between these times. We also denote by $\ell_r \leq |I_r|$ the number of items from I_r that are in the algorithm's buffer when the algorithm decides to remove I_r (starting at time slot t_r).

Claim 12. *For every batch (I,j) with m integral segments, and for every $1 \leq p < m$, we have that $\Delta_p \geq \sum_{r=p+1}^{m-1}\ell_r$.*

Proof: Notice that for every $1 \leq r < m$, $\Delta_r > 0$. Otherwise the time slot where the algorithm starts removing the items in I_r is after where they are matched by the batch (I,j). Thus, the algorithm removes all the remaining items of I. This mean that there would be no more integral segments after I_r.

We now show that given $1 \leq r < m-1$, we have that $\Delta_{r+1} < \Delta_r - \ell_r$. At time $t_r + |I_r|$ when we reach past the end of I_r's eviction, there are no items of this color in B. Therefore, all the items of the following fractional segment (denoted by F_{r+1}), and all the integral items that were in B at the time that segment I_{r+1} was sheduled (starting at time slot t_{r+1}), enter the buffer during the input interval $[t_r+|I_r|+1, t_{r+1}]$. Therefore, $t_{r+1} > t_r+|I_r|+|F_{r+1}|+\ell_{r+1}$. Combined with the fact that, $j_{r+1} = j_r + |I_r| + |F_{r+1}|$, we get that $\Delta_{r+1} = j_{r+1} - t_{r+1} < j_r + |I_r| + |F_{r+1}| - (t_r + |I_r| + |F_{r+1}| + \ell_{r+1}) = \Delta_r - \ell_{r+1}$ This complets the proof as $\Delta_p \geq \Delta_{p+1} + \ell_{p+1} \geq \Delta_{p+2} + \ell_{p+2} + \ell_{p+1} \geq \cdots \geq \Delta_{m-1} + \sum_{p<r<m}\ell_r \geq \sum_{p<r<m}\ell_r$. ∎

Claim 13. *Every batch (I,j) such that the first item $i \in I$ is in B at time j contains a constant number of segments.*

Proof: Let m be the number of integral segments in (I,j). We start by showing that $\Delta_1 < \frac{11k}{100\ln k}$. We assume that the first segment of I is a fractional segment. Otherwise, as the first item $i \in I$ is in B at time j, all of I is a single integral segment. Let F_1 be the first fractional segment. We also assume that $|F_1| > \frac{11k}{100\ln k}$ as otherwise $\Delta_1 < \frac{11k}{100\ln k}$ is triviall, because $t_1 > j$ and $j_1 = j + |F_1|$. As there can be at most $\frac{11k}{100\ln k}$ fractional items at time j, at least $|F_1| - \frac{11k}{100\ln k}$ items are added to F_1 after time j. Furthermore, there are at least $\frac{k}{100\ln k}$ items from I_1 that arrive before time slot t_1. These items arrive after the items of F_1 and therefore after time slot j. Therefore, $t_1 \geq j + |F_1| - \frac{11k}{100\ln k} + \frac{k}{100\ln k} \geq j_1 - \frac{k}{10\ln k}$. Thus, in this case $\Delta_1 \leq \frac{k}{10\ln k}$.

We now use Claim 12 and get that

$$\Delta_1 \geq \sum_{1<r<m}\ell_r \geq (m-2)\cdot\frac{k}{100\ln k}.$$

The upper and lower bounds on Δ_1 imply that $m < 13$. ∎

The proof of the following claim appears in the full version of the paper [9].

Claim 14. *For every batch (I,j) such that for every item $i \in I$ it holds that $i \notin B$ at the time it is scheduled by (I,j),*

$$\sum_{i\in I}(\hat{y}_i + \bar{y}_i) - \sum_{t=j}^{j+|I|-1}\hat{z}_t = O(\log\log k).$$

We are now ready to prove the main lemma.

Lemma 15. *The dual solution (y,z) is a feasible solution of* $\text{LP}_{k'}$.

Proof: Consider a dual constraint indexed (I,j). We partion the pseudo-dual cost $\hat{\sigma}_{I,j} + \sum_{i\in I}\bar{y}_i$ of (I,j) into two parts. Let $i \in I$ be the first item for which $i \in B$ at the time it is matched by (I,j). Partition (I,j) into two sub-batches (I_1,j), (I_2,j') such that I_1 contains all the items in I smaller than i, I_2 contains the rest of I's items, and $j' = M_{I,j}(i)$. From Claim 14 the pseudo-dual cost of (I_1,j) is $O(\log\log k)$. Any integral segment of (I_2,j') contributes exactly $\frac{1}{2}$ to $\sum_{i\in I_2}\bar{y}_i$. Only the last integral segment can have a positive contribution to $\hat{\sigma}_{I_2,j'}$, as any integral block with positive contribution to a batch (I,j) evicts all the remaining items of I. Any fractional segment contributes at most $O(\log\log k)$ to $\hat{\sigma}_{I_2,j'}$, by Claim 11. Therefore, as the

total number of segments in (I_2, j') is bounded by an absolute constant (Claim 13), the total pseudo-dual cost of (I_2, j') is also $O(\log\log k)$. We therefore conclude that the dual solution (y, z), derived by scaling down $(\hat{y} + \bar{y}, \hat{z})$ by an appropriate factor of $O(\log\log k)$, is feasible. ∎

B. Bounding the primal cost

Here we bound the cost of the primal solution using the cost of the dual solution.

Lemma 16. *At the end,*

$$\sum_{(I,j)} x_{I,j} = O(1) \cdot \left(\sum_{i=1}^{n} \hat{y}_i + \sum_{i=1}^{n} \bar{y}_i - \sum_{j=k'+1}^{k'+n} \hat{z}_j \right).$$

Proof: We partition the primal cost of the algorithm into three parts, according to the reason for incurring the cost.

Part 1 (regular execution): Consider an increase $d\mu$ in μ during regular execution, and let t be the current output time slot. By the definition of the algorithm, the dual variables that are raised at time μ are $\{\hat{y}_i\}_{i\in B}$, $\{\hat{y}_i\}_{t\le i\le n}$, and $\{\hat{z}_j\}_{t\le j\le k'+n}$. Therefore,

$$\frac{d\left(\sum_{i=1}^{n} \hat{y}_i - \sum_{j=k'+1}^{k'+n} \hat{z}_j\right)}{d\mu} = |B| + n - t + 1 - (k' + n - t + 1) = |B| - k'.$$

Let x, $\hat{x}$ be the (partial) primal and pseudo-primal solutions at time μ. For every color c for which $B_c^{act} \neq \emptyset$ is fractional, let (I_c, j_c) be the batch of color c that maximizes $\frac{d\hat{x}_{I,j}}{d\mu}$ (i.e., at time μ the rate of increase of $\hat{x}_{I,j}$ dictates the rate at which color c is removed from the buffer). Notice that during regular execusion, if $x_{I,j}$ increases then $B_{c(I)}^{act}$ must be fractional. Thus, by definition,

$$\begin{aligned}
\frac{d\sum_{(I,j)} x_{I,j}}{d\mu} &= \sum_{\text{fractional } c} \frac{d\hat{x}_{I_c,j_c}}{d\mu} \\
&= \sum_{c:\ \hat{\sigma}_{I_c,j_c}<1} \frac{1}{\ln k} \cdot \frac{d\hat{\sigma}_{I_c,j_c}}{d\mu} + \sum_{c:\ \hat{\sigma}_{I_c,j_c}\ge 1} \frac{d\hat{\sigma}_{I_c,j_c}}{d\mu} \cdot \hat{x}_{I_c,j_c} \\
&\le \sum_{c:\ \hat{\sigma}_{I_c,j_c}<1} \frac{|B_c^{act} \cap I_c|}{\ln k} + \sum_{c:\ \hat{\sigma}_{I_c,j_c}\ge 1} |B_c^{act} \cap I_c| \cdot \hat{x}_{I_c,j_c},
\end{aligned}$$

We bound the first term as follows:

$$\sum_{c:\ \hat{\sigma}_{I_c,j_c}<1} \frac{|B_c^{act} \cap I_c|}{\ln k} \le \frac{|B|}{\ln k} \le |B| - k'.$$

The last inequality follows from the fact that the volume in the buffer of items in B is at least $k - \frac{12k}{100\ln k}$ (an immediate consequence of Corollary 8). Because the volume in the buffer of items in B is at least $k - \frac{12k}{100\ln k}$ also $|B| \ge k - \frac{12k}{100\ln k}$. So,

$$\begin{aligned}
|B| - k' &= |B| - (k - \frac{12k}{100\ln k}) + (k - \frac{12k}{100\ln k}) \\
&\quad -(k - \frac{2k}{\ln k}) \\
&> |B| - (k - \frac{12k}{100\ln k}) + \frac{k}{\ln k} \\
&> \frac{|B| - (\frac{12k}{100\ln k}) + k}{\ln k} > \frac{|B|}{\ln k}.
\end{aligned}$$

We now bound the second term $\sum_{c:\hat{\sigma}_{I_c,j_c}\ge 1} |B_c^{act} \cap I_c| \cdot \hat{x}_{I_c,j_c}$. We show that for every color c,

$$|B_c^{act} \cap I_c| \cdot \hat{x}_{I_c,j_c} \le O(1) \cdot \sum_{(I,j)} (x_{I,j} \cdot |B_c \cap I|). \quad (4)$$

The difficulty in proving this is the following. Any increase in $\hat{x}_{I_c,j_c}$ lower bounds the increase in $x_{I,j}$, if the batch (I, j) is relevant to (I_c, j_c) at that time. In this case, it is possible to extend the scheduled batch (I, j) to include all the items in $B_c \cap I_c$. However, the batch might terminate before removing all those items because it reaches an item that is in B_c^{frz} at the time it needs to be scheduled. The regular reset of $\hat{x}_{I_c,j_c}$ takes care of this problem, as we show below.

In order to show Inequality (4), we consider three cases, according to the current value of μ. The first case is when $\mu < \mu_0(I_c, j_c)$ (i.e., before $\hat{x}_{I_c,j_c}$ experiences a regular reset). Notice that every batch (I, j) that is relevant to (I_c, j_c) and is scheduled starting at some time $\mu' \le \mu$, removes more than half of the items in $B_c(\mu)$, because at the current μ less than half of $B_c(\mu)$ arrived after the first item $f = f(I_c, j_c)$ that causes any interruption in a relevant scheduled batch. In this case,

$$\begin{aligned}
|B_c^{act}(\mu) \cap I_c| \cdot \hat{x}_{I_c,j_c} &\le |B_c(\mu)| \cdot \hat{x}_{I_c,j_c} \\
&\le |B_c(\mu)| \cdot \sum_{\text{relevent } (I,j)} x_{I,j} \\
&\le 2 \cdot \sum_{(I,j)} |B_c(\mu) \cap I| \cdot x_{I,j}.
\end{aligned}$$

The second case is when $\mu_0(I_c, j_c) \le \mu < \mu_1(I_c, j_c)$, where $\mu_1(I_c, j_c)$ is the time at which f is scheduled to be removed completely from the buffer. Notice that $\hat{x}_{I_c,j_c}$ is reset at $\mu_0 = \mu_0(I_c, j_c)$, so $\hat{x}_{I_c,j_c} \le \sum_{(I,j)} x_{I,j}$, where the sum is taken over relevant (I, j) that are scheduled after time μ_0. Consider such (I, j). If (I, j) is never interrupted (something that might happen if $B_c^{frz} \neq \emptyset$ at the time we reach the end of I), then clearly $B_c^{act}(\mu) \cap I_c \subseteq |B_c(\mu) \cap I|$. Otherwise, let $\mu' = \mu'_{I,j}$ denote any point in the time interval where (I, j) was scheduled (the sets don't change during that interval). Less than half the items in $B_c(\mu_0)$ arrived before f, so this remains true also for $B_c(\mu')$. As $\mu < \mu_1(I_c, j_c)$, we have that $f \in B_c(\mu)$. Set $\mu'' = \mu''_{I,j}$ to be the minimum time in $[\mu', \mu]$ when $B_c^{frz}(\mu'') \neq \emptyset$. If no such time exists, set $\mu'' = \mu$. Clearly, $|B_c^{act}(\mu'')| > \frac{10}{11}|B_c(\mu'')|$. Let $F = \{f, f+1, f+2, \ldots, n\}$ (i.e., the input items starting with f). Now, $|B_c(\mu'') \cap F| \ge \frac{1}{2} \cdot |B_c(\mu'')|$, so

$$|B_c^{act}(\mu'') \cap F| \ge \left(\frac{1}{2} - \frac{1}{11}\right) \cdot |B_c(\mu'')| > \frac{2}{5} \cdot |B_c(\mu'')|.$$

Clearly, (I, j) schedules all of $B_c^{act}(\mu'')$. Notice that $B_c^{act}(\mu'') \cap F \subseteq B_c(\mu)$ and $|B_c(\mu)| < \frac{12}{10} \cdot |B_c(\mu'')|$.

Combining everything together,

$$\begin{aligned}
|B_c^{act}(\mu) \cap I_c| \cdot \hat{x}_{I_c,j_c} &\leq |B_c(\mu)| \cdot \hat{x}_{I_c,j_c} \\
&\leq \sum_{(I,j)} |B_c(\mu)| \cdot x_{I,j} \\
&< \sum_{(I,j)} \frac{12}{10} \cdot |B_c(\mu''_{I,j})| \cdot x_{I,j} \\
&\leq \sum_{(I,j)} 3 \cdot |B_c^{act}(\mu''_{I,j}) \cap F| \cdot x_{I,j} \\
&\leq 3 \cdot \sum_{(I,j)} |B_c(\mu) \cap I| \cdot x_{I,j}.
\end{aligned}$$

The last case is when $\mu \geq \mu_1(I_c, j_c)$. In this case, consider all the scheduled batches that include f. Their total weight is 1, and they've all been scheduled before the current μ. Because f interrupted a relevant batch, all these batches must be relevant. A weight of less than $\frac{1}{10}$ of these batches is interrupted before time μ, otherwise we would have executed Case 6, removing all the remaining items of I_c. This contradicts the definition of (I_c, j_c) as the batch that currently, at time μ, controls the rate at which color c is evicted from the buffer. Thus, a weight of at least $\frac{9}{10}$ of the scheduled batches that include f schedules at time μ all the items in $B_c^{act}(\mu) \cap I_c$. On the other hand, by Claim 10, $\hat{x}_{I_c,j_c} \leq \frac{11}{10}$. Therefore,

$$\begin{aligned}
|B_c^{act}(\mu) \cap I_c| \cdot \hat{x}_{I_c,j_c} &\leq \frac{11}{9} \cdot \sum_{(I,j)} |B_c^{act}(\mu) \cap I| \cdot x_{I,j} \\
&\leq \frac{11}{9} \cdot \sum_{(I,j)} |B_c(\mu) \cap I| \cdot x_{I,j}.
\end{aligned}$$

Therefore, regardless of the value of the current time μ,

$$\begin{aligned}
&\sum_{c:\hat{\sigma}_{I_c,j_c} \geq 1} |B_c^{act} \cap I_c| \cdot \hat{x}_{I_c,j_c} \leq \\
&\quad 3 \cdot \sum_{c:\hat{\sigma}_{I_c,j_c} \geq 1} \sum_{(I,j)} |B_c \cap I| \cdot x_{I,j} \leq 3 \cdot (|B| - k'),
\end{aligned}$$

where the last inequality follows from Claim 9. Thus, summing the bounds on the two terms,

$$\frac{d \sum_{(I,j)} x_{I,j}}{d\mu} \leq 4 \cdot (|B| - k') \leq 4 \cdot \frac{d\left(\sum_{i=1}^{n} \hat{y}_i - \sum_{j=k'+1}^{k'+n} \hat{z}_j\right)}{d\mu},$$

which implies trivially that the total primal cost due to regular execution of the algorithm is at most 4 times the dual cost.

Part 2 (Case 3 and Case 5 execution): Each time an integral block is evicted (Case 5), $\sum_{i=1}^{n} \bar{y}_i$ is raised by $\frac{1}{2}$. Preceding each such eviction there is a specific Case 3 execution, when the block became integral. These Case 3 and Case 5 executions incur together a primal cost of at most 3. (Case 3 evicts a color from the buffer at a cost of at most 1. Case 5 schedules an intergal block, and may suspend fractional batches of toal weight 1. So the cost of Case 5 is at most 2.) Therefore, the total primal increment due to Case 3 and Case 5 is at most $6 \cdot \sum_{i=1}^{n} \bar{y}_i$.

Part 3 (Case 6 execution): Case 6 costs at most 2 (just like Case 5). Each time we execute Case 6 on color c, we've moved past the end of regular execution fractional scheduled batches of color c with total weight at least $\frac{1}{10}$. After the end of this eviction, B does not contain any color c items, therefore the next Case 6 execution is due to distinct fractional scheduled batches. Therefore the primal increase as a result of Case 6 is at most 20 times the primal increase due to regular executions. By the above analysis of regular execution, this incurs a cost of at most 80 times the dual cost. ■

REFERENCES

[1] A. Aboud. Correlation clustering with penalties and approximating the reordering buffer management problem. Master's thesis, Computer Science Department, The Technion - Israel Institute of Technology, January 2008.

[2] A. Adamaszek, A. Czumaj, M. Englert, and H. Räcke. Almost tight bounds for reordering buffer management. In *Proc. of the 43rd Ann. ACM Symp. on Theory of Computing*, pages 607–616, June 2011.

[3] A. Adamaszek, A. Czumaj, M. Englert, and H. Räcke. Optimal online buffer scheduling for block devices. In *Proc. of the 44th Ann. ACM Symp. on Theory of Computing*, pages 589–598, June 2012.

[4] N. Alon, B. Awerbuch, Y. Azar, N. Buchbinder, and J. Naor. The online set cover problem. *SIAM J. Comput.*, 39(2):361–370, 2009.

[5] S. Arora, E. Hazan, and S. Kale. The multiplicative weights update method: a meta-algorithm and applications. *Theory of Computing*, 8(1):121–164, 2012.

[6] Y. Asahiro, K. Kawahara, and E. Miyano. NP-hardness of the sorting buffer problem on the uniform metric. Unpublished, 2010.

[7] N. Avigdor-Elgrabli and Y. Rabani. An improved competitive algorithm for reordering buffer management. In *Proc. of the 21st Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 13–21, January 2010.

[8] N. Avigdor-Elgrabli and Y. Rabani. A constant factor approximation algorithm for reordering buffer management. In *Proc. of the 24th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 973–984, January 2013.

[9] N. Avigdor-Elgrabli and Y. Rabani. An optimal randomized online algorithm for reordering buffer management. `arXiv:1303.3386 [cs.DS]`, March 2013.

[10] Y. Azar, U. Bhaskar, L. Fleischer, and D. Panigrahi. Online mixed packing and covering. In *Proc. of the 24th Ann. ACM-SIAM Symp. on Discrete Algorithms*, pages 85–100, January 2013.

[11] N. Bansal, N. Buchbinder, A. Madry, and J. Naor. A polylogarithmic-competitive algorithm for the k-server problem. In *Proc. of the 52nd Ann. IEEE Symp. on Foundations of Computer Science*, pages 267–276, October 2011.

[12] N. Bansal, N. Buchbinder, and J. Naor. A primal-dual randomized algorithm for weighted paging. *J. ACM*, 59(4) (Article 19), 2012.

[13] D. Blandford and G. Blelloch. Index compression through document reordering. In *Data Compression Conference*, pages 342–351, 2002.

[14] N. Buchbinder and J. Naor. The design of competitive online algorithms via a primal-dual approach. *Found. Trends Theor. Comput. Sci.*, 3(2–3):93–263, February 2009.

[15] H.-L. Chan, N. Megow, R. van Stee, and R. Sitters. A note on sorting buffers offline. *Theor. Comput. Sci.*, 423:11–18, 2012.

[16] M. Englert, H. Röglin, and M. Westermann. Evaluation of online strategies for reordering buffers. *ACM Journal of Experimental Algorithmics*, 14 (Article 3), 2009.

[17] M. Englert and M. Westermann. Reordering buffer management for non-uniform cost models. In *Proc. of the 32nd Ann. Int'l Colloq. on Algorithms, Langauages, and Programming*, pages 627–638, 2005.

[18] K. Gutenschwager, S. Spiekermann, and S. Vos. A sequential ordering problem in automotive paint shops. *Int'l J. of Production Research, 42(9):1865–1878*, 2004.

[19] J. Krokowski, H. Räcke, C. Sohler, and M. Westermann. Reducing state changes with a pipeline buffer. In *Proc. of the 9th Int'l Workshop on Vision, Modeling and Visualization*, page 217, 2004.

[20] H. Räcke, C. Sohler, and M. Westermann. Online scheduling for sorting buffers. In *Proc. of the 10th Ann. European Symp. on Algorithms*, pages 820–832, 2002.