On the Randomized Competitive Ratio of Reordering Buffer Management with Non-Uniform Costs

Noa Avigdor-Elgrabli[1], Sungjin Im[2⋆], Benjamin Moseley[3], and Yuval Rabani[4]

[1] Yahoo! Labs Haifa, MATAM, Haifa 31095, Israel. noaa@yahoo-inc.com
[2] University of California, Merced, CA 95344. sim3@ucmerced.edu
[3] Washington University in St. Louis, MO, 63130. bmoseley@wustl.edu
[4] The Hebrew University of Jerusalem, Jerusalem 91904, Israel. yrabani@cs.huji.ac.il.

Abstract. Reordering buffer management (RBM) is an elegant theoretical model that captures the tradeoff between buffer size and switching costs for a variety of reordering/sequencing problems. In this problem, colored items arrive over time, and are placed in a buffer of size k. When the buffer becomes full, an item must be removed from the buffer. A penalty cost is incurred each time the sequence of removed items switches colors. In the non-uniform cost model, there is a weight w_c associated with each color c, and the cost of switching to color c is w_c. The goal is to minimize the total cost of the output sequence, using the buffer to rearrange the input sequence.
Recently, a randomized $O(\log\log k)$-competitive online algorithm was given for the case that all colors have the same weight (FOCS 2013). This is an exponential improvement over the nearly tight bound of $O(\sqrt{\log k})$ on the deterministic competitive ratio of that version of the problem (Adamaszek et al. , STOC 2011). In this paper, we give an $O((\log\log k\gamma)^2)$-competitive algorithm for the non-uniform case, where γ is the ratio of the maximum to minimum color weight. Our work demonstrates that randomness can achieve exponential improvement in the competitive ratio even for the non-uniform case.

1 Introduction

Motivation and background. In the reordering buffer management problem (RBM) a stream of colored items enters a buffer of limited capacity k, which is used to permute the input stream. Once the buffer is full, any item can be removed from the buffer to the permuted output stream to make room for the next input item. This is repeated until the buffer is empty. The goal is to minimize the context switching cost of the output stream due to color changes. The literature considers various cost models. The simplest version is the uniform cost model, where each color switch costs 1. In this paper, we are concerned with the

⋆ Supported in part by NSF grant CCF-1409130.

so-called non-uniform cost model, where each color c has a weight w_c, and a switch in the output stream to color c costs w_c. In the online version of the problem, the decision on which item to remove from the buffer must be made on-the-fly without knowing the future input stream. In the offline version of the problem, the entire input stream is known in advance.

RBM models a wide range of applications in production engineering, logistics, computer systems, network optimization, and information retrieval (see, e.g., [6,15,14,12]). In essence, RBM, introduced in [15], gives a nice theoretical framework which allows us to study the tradeoff between buffer size and context switching costs. This tradeoff is evident in many applications. From the perspective of the theory of algorithms, this seemingly simple problem is NP-hard [8], and it presents significant algorithmic challenges both in the offline and the online settings. For instance, simple algorithms such as greedy or FIFO are known to have poor performance.

RBM was studied mostly in the online setting [15,10,11,9,3,1,5]. The performance guarantees for uniform RBM were essentially resolved in a sequence of papers. There is a deterministic $O(\sqrt{\log k})$-competitive online algorithm and a nearly matching $\Omega(\sqrt{\log k/\log\log k})$ lower bound [1]. The randomized competitive ratio is $\Theta(\log\log k)$. The lower bound is from [1] and the upper bound was recently proved in [5]. So, similar to some other online problems such as paging, randomness gives an exponential improvement in the competitive ratio. In the offline setting, there is an $O(1)$-approximation algorithm [4] (see alternative algorithms in [5,13]), but no hardness of approximation result beyond NP-hardness of the exact solution.

In contrast, there is a wide gap in our understanding of non-uniform RBM. The best known upper bound on the competitive ratio of non-uniform RBM is $\min\{\log k/\log\log k, \sqrt{\log k\gamma}\}$, where γ is the ratio of maximum to minimum color weight. This bound combines the results for two deterministic algorithms from [3] and [1]. For γ which is polynomial in k, the algorithm in [1] nearly matches the deterministic lower bound for the uniform case. The above-mentioned uniform case upper bounds of $O(\log\log k)$ on the randomized competitive ratio and of $O(1)$ on the approximation guarantee seem to use uniformity inherently. So the randomized competitive ratio and the approximability of non-uniform RBM were far from settled. Very recently, an offline approximation guarantee of $O(\log\log k\gamma)$ was shown [13]. Hence, the looming question concerning non-uniform RBM was if randomness can give an exponential improvement of the competitive ratio as it did for the uniform case. (We note that in the case of paging, for instance, the analogous question regarding the randomized competitive ratio of weighted caching remained open for a very long time.)

Our results. In this paper, we answer the above question in the affirmative. Specifically, we prove the following theorem.

Theorem 1. *There is a randomized $O((\log\log k\gamma)^2)$-competitive online algorithm for the non-uniform RBM problem.*

Our algorithm is based on the online primal-dual schema (see [7] for a survey). The algorithm consists of two phases. In the first phase, the algorithm computes deterministically a feasible fractional solution to an LP relaxation for non-uniform RBM. The LP solution is computed online. In parallel, the algorithm examines the partial LP solution and rounds it online using randomness to get an integral RBM solution which is the output of the algorithm. We lose a factor of $O(\log\log k\gamma)$ in each of the two phases. Interestingly, both phases use a resource augmentation argument to bound the cost of the online solution they produce. In the first phase, the cost of the online generated LP solution is compared against the cost of a dual LP solution for a smaller buffer (see [10,2,5] for previous application of this idea in similar contexts). In the second phase, resource augmentation is used to give the integral solution a bit more buffer space than the LP solution that is rounded to generate it. Overall, the integral solution uses a buffer of size k (thus respecting the buffer capacity constraint), the LP primal solution uses a buffer of size $k - \frac{k}{\log k\gamma}$, and the LP dual that is used for bounding the LP cost is for a buffer of size roughly $k - \frac{2k}{\log k\gamma}$. We note that all the previous resource augmentation proofs for RBM either did not apply to the non-uniform case, or they did not prove sufficiently tight bounds. Our proof is new and different from previous proofs.

The first phase of computing the LP solution is generally framed after the algorithm for the uniform case in [5]. The algorithm combines two methods. One method uses the online version of the multiplicative weights update method (see [7]) and works well as long as the color blocks in the buffer do not exceed a size of $O(k/\log k\gamma)$. The other method uses an integral dual fitting-based algorithm that works well when all the color blocks in the buffer have size $\Omega(k/\log k\gamma)$ when they are removed. In [5], the main difficulty was to combine the two algorithms to work well when the buffer contains a mixture of the two types of color blocks. However, the way the two algorithms were combined in [5] inherently uses uniformity, because whenever there was a switch between the two types in one color, other completely arbitrary colors could be charged. In order to facilitate the combination in the non-uniform case, the algorithm and its analysis had to be modified. The result happens to be a simpler and cleaner algorithm and analysis.

The second phase of the algorithm is motivated by the recent offline approximation algorithm in [13]. There, a solution to a slightly different LP was rounded to give an $O(\log\log k\gamma)$ approximation guarantee (without using resource augmentation). However, the algorithm in [13] had several steps that rely crucially on offline information about the LP solution. In particular, that algorithm makes decisions based on when the LP removes certain items in the future. Here we show how to round an LP solution without using future information, exploiting resource augmentation instead. The algorithm is substantially different, simpler than the offline rounding algorithm, and even simpler than the rounding algorithms for the uniform case. (The uniform case rounding algorithms relied crucially on uniformity, and it does not seem that they could be modified to handle color weights.)

Due to lack of space, most of the proofs are deferred to the full version of the paper. We give some informal intuition on the analysis.

2 Preliminaries

In the reordering buffer management problem we consider, there is a sequence $\mathcal{I}$ of n items that arrive over time online. Each item i is associated with a specific color $c(i)$ which stands for the item's type. A single item arrives at every time step from 1 to n and we assume items are indexed in increasing order. Each color c has a positive weight w_c and we denote the ratio of maximum to minimum weight by γ. There is a buffer of size k, and we are allowed to hold items up to the buffer size. Once the buffer becomes full, we are forced to output an item. The goal is to reorder the items using the buffer to minimize the total cost of color switches in the output. Each color switch costs the weight of the color switched to.

Another useful view of the output is to view the sequence of items output as a partition of items into color blocks – a color block or simply block refers to a sequence of items of the same color. In this view, each block of color c contributes w_c to the objective. We assume without loss of generality that each block I is a contiguous sequence of items of the same color ordered in first-in-first-out manner starting with the first arriving item in I. Let $c(I)$ denote the color of the items in I. When a block I is associated with the time t that its first item is removed from the buffer, we call the pair (I, t) a *batch*. For a batch $b = (I, t)$ and an item $i \in I$, we denote by $M_b(i)$ the time that i is removed from the buffer. Note that the total number of all possible blocks is polynomial in n, and so is the total number of possible batches.

For a given input instance, we let OPT_k denote the optimal solution with a buffer of size k. Throughout the analysis, we will compare an algorithm with a buffer of size k to an optimal or linear program solution with a buffer of size smaller than k. This will be clearly indicated when we are making the comparison. We appeal to the following theorem when comparing against a solution with a smaller buffer size. A similar theorem was shown for the unweighted version of the problem and we extend this to the weighted version. In our analysis, we will set k' to be roughly $k - \frac{k}{\log k\gamma}$, which can increase the cost of the optimal solution by at most a constant factor.

Theorem 2. *For any input sequence and $k' < k$, respectively,* $\mathrm{OPT}_{k'} \le O(1) \cdot (\frac{k}{k'} + (k-k')\frac{\log k'\gamma}{k'})\mathrm{OPT}_k$*, where* OPT_s *denotes the cost of the optimal solution using a buffer of size s.*

We use the following linear programming relaxation for the problem, which is defined over $x \ge 0$. It is similar to the relaxation introduced in [3].

$$\min \sum_{I,j} w_{c(I)} x_{I,j} \quad \text{s.t.} \sum_{(I,j), i \in I} x_{I,j} \ge 1 \qquad \forall i = 1, 2, \ldots, n \tag{1}$$

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

The quantity $x_{I,j}$, which we call the *height* of batch (I, j), refers to the amount by which the batch (I, j) is scheduled. It is an easy exercise to see this is a valid LP relaxation. The first constraint ensures that each item is processed by an amount of 1. The second constraint ensures that the total height of the intervals at a time step is at most 1. Put $\beta_{i,j} = \sum x_{I,j'}$, where the sum is taken over batches (I, j') such that $i \in I$ and $M_{I,j'}(i) \leq j$. So $\beta_{i,j}$ denotes the total amount item i is processed by time j. Also put $v_{i,j} = 1 - \beta_{i,j}$; this is the remaining "volume" of item i that still needs to be processed at time j. The dual of the linear program is over $y, z \geq 0$ and is given as follows.

$$\max \sum_{i=1}^{n} y_i - \sum_{j=k+1}^{k+n} z_j \quad \text{s.t.} \sum_{i \in I} y_i - \sum_{j'=j}^{j+|I|-1} z_{j'} \leq w_{c(I)} \qquad \forall (I, j) \tag{3}$$

We will denote the LP for a buffer of size k as LP_k and the dual for a buffer of size k as DP_k.

Our online algorithm will use this LP to guide its decisions. In particular, the algorithm approximately solves this LP in an online fashion. The algorithm simultaneously rounds this LP online to construct the solution. Formally the following is what we mean by solving the LP online. Consider any fixed time t. All batches considered so far end no later than time t – at the next time step $t + 1$, some of batches reaching this time moment t can be extended to time $t + 1$ by adding an extra element to the batch if there is an available element of the same color to be scheduled. In this case, the height of such a batch must remain the same. Note that formally, the batch changes to a larger batch. Also, new batches can start at the current time, and Constraints (2) must be satisfied at each time until time t. Finally, Constraints (1) must be eventually satisfied. Our algorithm and analysis are split into two parts. In Section 3 we show how to construct the LP solution online and in Section 4 we show how to round the LP solution in an online fashion.

3 Solving the LP Online

3.1 The algorithm

We give an online algorithm that constructs a primal fractional LP solution x for a buffer of size k. We prove that the cost of x, which is $\sum_{(I,j)} w_{c(I)} \cdot x_{I,j}$ is at most $O(\log \log(k\gamma))$ times the optimal cost. In order to prove this bound, the algorithm also constructs a dual solution (y, z) for a buffer of a smaller size $k' = k - \frac{k}{2 \ln(k\gamma)}$. The bound is then obtained by comparing the costs of the primal and dual solutions. The construction of (y, z) is done by scaling an infeasible solution $(\hat{y} + \bar{y}, \hat{z})$, where $(\hat{y}, \hat{z})$ is generated through a version of the online primal-dual schema, and $\bar{y}$ is an extra penalty imposed via a dual fitting procedure. Informally, $(\hat{y}, \hat{z})$ pays for removing from the buffer "small" blocks, and $\bar{y}$ pays for removing "large" blocks. The meaning of "small" and "large" will be made precise in the discussion below. In addition to all of the above variables, we also maintain pseudo-primal variables $\tilde{x}$ that will help us construct the fractional solution x.

The algorithm proceeds as follows. Initially, all primal and dual variables are set to 0 (this includes x, y, z, $\tilde{x}$, $\hat{y}$, $\hat{z}$, $\bar{y}$). Our initial output slot is $t = k+1$, and the first k input items are fully in the buffer. We raise some of the dual variables at a uniform rate, so it is convenient to think about the solution as a function of $\mu \in [0, \infty)$, where $\mu = 0$ denotes the initial state. (Of course, the implementation is not a continuous process—there is a finite sequence of "interesting" values of μ where something happens, and the algorithm can compute those thresholds. However, it is convenient to describe the continuous process.)

The algorithm increases all the variables $\hat{y}_i$ for all input items i in that are in the buffer and have not been scheduled to be removed completely from the buffer (see below), and all the variables $\hat{z}_j$ for all output slots $j \geq t$ at the same rate $d\mu$. Notice that this affects future i-s and j-s. We don't need their values until we reach them, and at that point the value can be computed given the past. Raising some of the variables in $(\hat{y}, \hat{z})$ changes the primal solution x. In order to see how this is done, consider the buffer's contents. Of the total volume of k, there might be some volume that we already decided to remove, but its removal will happen past the current output slot t. We'll call it *phantom volume* and the rest *real volume*. Part of the real buffer volume is kept as *frozen volume* (it will consist only of integral items). We'll call the real volume that is not frozen *active volume*.

Consider a dual constraint indexed (I, j). Put $\sigma_{I,j} = \sum_{i \in I} \hat{y}_i - \sum_{j'=j}^{j+|I|-1} \hat{z}_{j'}$. This is the current *dual cost* of the batch (I, j). Notice that we know the current dual cost even if the batch is matched to output slots we haven't yet reached (and even if it includes items we haven't yet seen). As we raise $(\hat{y}, \hat{z})$, the dual cost of some of the batches may increase. We want to measure only part of this increase, the part that is due to items that contribute to the active volume in the buffer. We call this part the *pseudo-dual cost* and we denote it by $\tilde{\sigma}_{I,j}$. In order to explain this, notice that $\frac{d\sigma_{I,j}}{d\mu}$ is precisely the number of items of color $c(I)$ that contribute to the real volume and are scheduled by (I, j) before the current output slot t. Thus, we raise $\tilde{\sigma}_{I,j}$ at a rate $\frac{d\tilde{\sigma}_{I,j}}{d\mu}$ which is the number of items of color $c(I)$ that contribute to the active volume (i.e., excluding items that are frozen) and are scheduled by (I, j) before the current output slot t. This is what normally happens with $\tilde{\sigma}$. However, there are special "events" that trigger a reset of $\tilde{\sigma}_{I,j}$ to 0. After a reset, $\tilde{\sigma}_{I,j}$ grows again at the rate defined above.

The pseudo-dual costs determine the values of the pseudo-primal variables. We maintain at all times the equation

$$\tilde{x}_{I,j} = \begin{cases} \frac{1}{\ln(k\gamma)} \cdot \frac{\tilde{\sigma}_{I,j}}{w_{c(I)}} & \tilde{\sigma}_{I,j} < w_{c(I)}, \\ \frac{1}{\ln(k\gamma)} \cdot e^{\tilde{\sigma}_{I,j}/w_{c(I)} - 1} & \tilde{\sigma}_{I,j} \geq w_{c(I)}. \end{cases}$$

(It should be noted that when we reset $\tilde{\sigma}_{I,j}$ this also resets $\tilde{x}_{I,j}$. However, the reader will soon notice that by Equation (4) this does not reset any actual primal variable—such a reset would violate our intention to construct the solution online.) Now, the items that contribute to the active volume are further classified as *fractional* or *integral*. For each color present in the active volume there are either fractional or integral items (contributing to the active volume), but not both. We say that the active items of a specific color constitute an *active block* in the buffer, which is either a fractional active block or an integral active block.

We are now ready to explain how the schedule up to time $t-1$ is extended (i.e., how to update the primal solution x). It will be convenient to present the algorithm as choosing batches to schedule and then increasing their height continuously until some event stops the increase and sets the final height of the scheduled batch. Also, when we decide to schedule a batch, we may not know its full extent, because it may end with items that we haven't yet reached in the input stream. However, we will be able to extend the batch as we go along, so in describing the algorithm, we also specify the rule that determines the extent of the batch, and this rule is checked as we go along. Notice that the current output slot t might be already partially filled with previously scheduled batches that haven't reached their end (the partial schedule from t onward is precisely the phantom volume). So our goal is to fill up output slot t and then move on to the next output slot that is not completely filled up.

If an output slot gets filled up, or an item gets scheduled completely, this stops the increase of the height of the current batches, and we execute the following procedure, depending on the event.

Filling up an output slot: When we fill up the output slot t, we have to advance to a later output slot and start the extention process afresh. In this case, new items enter into the buffer, replacing the volume that is removed from the buffer in the filled up output slots (t and possibly later slots). When an item enters the buffer, it is usually frozen, unless the buffer contains an integral active block of this color. In the latter case, the item is sometimes appended to the integral block, according to the rule that specifies the end of the batches that will remove this block from the buffer. If the item is not appended to the integral block, it is frozen as usual.

Scheduling an item entirely: At some point, the initial items of some batch may get scheduled with total height 1. This means that they are either removed from the buffer, or (if they are scheduled in the future) they no longer contribute to the real volume (but they still contribute to the phantom volume). In this case the height of the relevant batch is fixed, and we may continue scheduling a new batch of this color that begins with the items that still contribute to the real volume.

We now describe how an output slot t gets filled up. There are a few cases to consider:

Evicting integral blocks: We first consider the integral active blocks. If there exists (I,j) for which $\tilde{x}_{I,j}$ reached 1 and the items of color $c(I)$ in the buffer are an integral active block B, we set $\bar{y}_i = \frac{w_{c(I)}}{2|B|}$ for all $i \in B$. We reset $\tilde{\sigma}_{I',j'}$ (and hence $\tilde{x}_{I',j'}$) for all (I',j') of color $c(I)$. Then, we schedule batches consisting of this block followed by all the items that can be appended to it assuming it is removed starting from output slot t. The total height of the batches we schedule is 1 (i.e., we remove the block and the appended items completely from the buffer), but we may have to split the height across several batches because some of the output volume beyond time $t-1$ might be already taken by previously scheduled batches.

Releasing frozen items: We next consider the frozen items in the buffer. We release frozen items in two cases. Firstly, if there is a color c with more than $\frac{k}{100\ln(k\gamma)}$ frozen items in the buffer, we first schedule batches to remove all the volume of the fractional active block of color c from the buffer (they all end with the same last unfrozen item of color c; notice that while we schedule these batches, t might move forward). Then, we reset $\tilde{\sigma}_{I,j}$ to 0 for all batches (I,j) with $c(I)=c$. Finally, we move the frozen items (including additional items that may have been added while removing the preceding fractional volume) to form an integral active block. Secondly, if there is a fractional active block with fewer than $\frac{k}{10\ln(k\gamma)}$ items, we add all the frozen items of this color to the fractional active block. Notice that this event can happen while we are filling up output slot t (because some items get scheduled completely).

Scheduling fractional blocks: We finally consider fractional active blocks (assuming none of the above cases can now be applied). We schedule them in batches in parallel. Such a batch (J,t) consists of the sequence of items in the fractional active block, followed by the items of this color that are in the fractional active block at the time that they are needed to continue the batch. Thus, a fractional batch ends in one of three cases: (i) we haven't reached the next input item of this color; (ii) the next input item of this color is frozen (in this case we say that the batch is *interrupted*); (iii) the next input item of this color begins an integral block. (Notice, that when a batch is being scheduled, we may know only a prefix of the sequence of items in the batch. However, we can extend this sequence on-the-fly and transfer the fractional weight from the prefix to the extended batch as we go along. This does not change the packing of the items in the past time slots, only in future time slots.)

All these fractional batches are scheduled in parallel. Their height is increased as μ grows by the following rate.

$$\frac{dx_{J,t}}{d\mu} = \max_{(I,j)} \left\{ \frac{d\tilde{x}_{I,j}}{d\mu} :\ c(I)=c(J) \right\}. \tag{4}$$

We increase their height until, as explained above, some event triggers a change in the batch or in t. A batch (J,t) is said to be *relevant to* (the dual cost of) (I,j) for every (I,j) that has at some point μ a positive value in the right-hand side of the above expression (i.e., $c(I)=c(J)$ and $\tilde{x}_{I,j}$ grows while $x_{J,t}$ grows).

Regular resets: Occasionally while scheduling fractional batches, we reset some $\tilde{\sigma}_{I,j}$ to 0. We will call this a *regular reset* (to distinguish it from other resets that happen while dealing with integral blocks). Suppose that a fractional batch (J,t) is interrupted at output slot $t'>t$. Let i be the interrupting item (i.e., i is frozen when we reach t'). We consider the set of batches that (J,t) is relevant to. For such a batch (I,j) we reset $\tilde{\sigma}_{I,j}$ if and when the following three conditions hold: (i) The block I contains i; (ii) item i is the first item of I that ever interrupted a batch that is relevant to (I, j); (iii) more than half of the items of color $c(i)$ that contribute to the real volume are frozen. Notice that for any (I, j), a regular reset happens at most once. We denote the value of μ at the time of this regular reset by $\mu_0(I, j)$ and the interrupting item i by $f(I, j)$. If (I, j) never experiences a regular reset, we put $\mu_0(I, j) = \infty$. Also recall that if $\tilde{\sigma}_{I,j}$ is reset to 0, automatically $\tilde{x}_{I,j}$ is reset to 0.

Occasional cleanup: We sometimes clean up the buffer of a color c. The condition for cleaning up color c is as follows: since the previous execution of this step, we just moved past the end of scheduled fractional batches of color c of total height at least $\frac{1}{10}$. (For this purpose we count only batches that are removed while μ increases and not batches that are removed during cleanup.) In this case, we append the frozen items of color c to the color c batches that occupy the current output slot. Then, if there are still items of color c that contribute to the real volume, we schedule additional batches to remove all color c items from the real volume. Obviously, all the frozen items of color c will now be part of the phantom volume.

3.2 Competitive analysis

Clearly, the algorithm computes a feasible primal solution x. We show that the primal cost of x (which uses a buffer of size k) is proportional to the dual cost of the infeasible solution $(\hat{y} + \bar{y}, \hat{z})$ (which uses a smaller buffer size k'). Then we prove an upper bound on the factor that is needed to scale $(\hat{y} + \bar{y}, \hat{z})$ to a feasible solution (y, z).

Properties of the primal solution. We begin with a bound on the phantom volume. This justifies the choice of k'.

Lemma 1. *At any time during the execution of the algorithm, the phantom volume never exceeds* $\frac{12k}{100\ln(k\gamma)}$.

Lemma 1 immediately implies the following corollary.

Corollary 1. *At any given time, the real volume in the buffer is more than* $k - \frac{12k}{100\ln(k\gamma)} \geq k'$.

Next we show that the pseudo-primal variables are bounded.

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

The main idea behind the proof is that $\tilde{x}_{I,j}$ is bounded by the total height of color $c(I)$ batches that are removed since the last reset of $\tilde{x}_{I,j}$. The total height of batches that extend beyond the current output slot is at most 1, and the total height of batches that ended is less than $\frac{1}{10}$, otherwise we would have executed a cleanup step.

Bounding the primal cost. We show that the primal cost of x is proportional to the dual cost of $(\hat{y}+\bar{y},\hat{z})$.

Lemma 2. *At the end,* $\sum_{(I,j)} x_{I,j} = O(1)\cdot\left(\sum_{i=1}^{n}\hat{y}_i + \sum_{i=1}^{n}\bar{y}_i - \sum_{j=k'+1}^{k'+n}\hat{z}_j\right)$.

The main idea of the proof is the following. We bound separately the cost of scheduling fractional blocks, the cost of evicting integral blocks, and the cost of cleanup. For fractional blocks, we relate the rate by which the primal cost is increased to the rate by which the dual cost is increased. We use the gap between the primal and dual buffer size and the fact that the real volume is most of the buffer (Corollary 1) to show that the dual cost increases sufficiently fast. For integral blocks, the increase in $\sum_{i=1}^{n}\bar{y}_i$ directly bounds the primal cost of evicting those blocks. The cleanup cost is charged against the primal cost of the fractional batches that caused the cleanup.

Dual feasibility. Here we show that if we scale $(\hat{y}+\bar{y},\hat{z})$ by a factor of $O(\log\log(k\gamma))$, then we get a feasible dual solution (x,y), namely, for every batch (I,j), $\sum_{i\in I} y_i - \sum_{j'=j}^{j+|I|-1} z_{j'} \le w_{c(I)}$. So fix a batch (I,j). The main idea of the proof is to partition I into *segments*, according to what the algorithm does with these items. A segment is a maximal substring of items that were all scheduled as a fractional block or an integral block. So there are alternating fractional and integral segments. (Notice that a fractional segment includes also items that were removed during cleanup.) We then partition (I,j) into two sub-batches (I_1,j), (I_2,j') as follows. Let $i\in I$ be the first item that still contributes to the real volume when the algorithm reaches the output slot that (I,j) matches to i. Then, I_1 contains all the items in I that precede i, and I_2 contains the rest of I's items (so $j' = M_{I,j}(i)$). The cost of each sub-batch is bounded using a different argument. Roughly speaking, in (I_1,j) the fractional segments do not incur a positive cost, and at most $O(\log\log(k\gamma))$ integral segments incur a positive cost of $O(w_{c(I)})$. In (I_2,j') there are $O(1)$ segments, and each fractional segment incurs a cost of $O(w_{c(I)})\cdot\log\log(k\gamma))$. This discussion leads to the following lemma.

Lemma 3. *The pair (y,z) is a feasible dual solution for a buffer of size k'.*

We conclude with the main result of this section.

Theorem 3. *The primal cost of the output x of the LP algorithm is within a factor of $O(\log\log(k\gamma))$ of the LP optimum.*

Proof. Notice that $\sum_{I,j} w_{c(I)}\cdot x_{I,j} \le O(1)\cdot\left(\sum_{i=1}^{n}\hat{y}_i + \sum_{i=1}^{n}\bar{y}_i - \sum_{j=k'+1}^{k'+n}\hat{z}_j\right) = O(\log\log(k\gamma))\cdot\left(\sum_{i=1}^{n} y_i - \sum_{j=k'+1}^{k'+n} z_j\right) \le O(\log\log(k\gamma))\cdot \mathrm{DP}_{k'} = O(\log\log(k\gamma))\cdot \mathrm{LP}_{k'} \le O(\log\log(k\gamma))\cdot \mathrm{LP}_k$. The first inequality uses Lemma 2. The second inequality uses Lemma 3. The third inequality uses Theorem 2.

4 Rounding the LP Online

In this section we give an algorithm that rounds the linear program solution of the previous section in an online fashion. Our online rounding requires a sampling which we name α-sampling. The α-sampling is essentially a "boosted-up" independent rounding. Let $0 < \alpha \leq 1$ be a constant to be fixed later. We sample each batch b starting at time t independently with probability $\min\{\alpha, x_b\}/\alpha$, and add it to a pool Bag. Define an item i's α-ready time, t_i^α as the first time t such that there is a batch $b \in \mathsf{Bag}$ that schedules i at time t – if no such batch b exists, then set $t_i^\alpha = \infty$. At any time $t \geq t_i^\alpha$, we say that i is α-ready at time t.

To see that we can do the sampling online, note that each batch in the LP keeps the same height from when it starts until it ends. Hence we can immediately decide whether to add a batch to Bag or not when the batch starts in the LP solution.

4.1 Online Rounding Algorithm

The online rounding algorihtms takes as input an online LP solution with a buffer of size $k' = k - \frac{k}{\log k\gamma}$ and returns an online algorithm using a buffer of size k. Recall that reducing the optimal solution's buffer by $\frac{k}{\log k\gamma}$ only increases its cost by a $O(1)$ factor, as we have shown in Lemma 2. In the previous section, we presented how to construct an LP solution online assuming the buffer size is k for notaitonal simplicity. The actual LP solution should has a buffer of size k' and the dual LP's buffer size should be scaled appropriately.

The algorithm at any time always outputs an item for the color that was previously output in the last time step if possible. Otherwise, the algorithm needs to decide which color to switch to. The algorithm has several rules on which color to switch to at time t and attempts to execute the rules in the following order. The first three rules are easy cases and the crux of the algorithm is the final two rules. The rules are similar to the algorithm in [13]. However, the algorithm in [13] required an additional rule and also the main rules in their algorithm used future offline information from the LP.

We require some notation to define formally the algorithm. Let ϵ be any constant between 0 and 1/100 and α be a constant at most ϵ. We will later set $\epsilon = 1/100$ and $\alpha = \epsilon$. Let $\mathcal{B}(t)$ denote (the set of items in) the algorithm's buffer at time t. Let $n_c^A(t)$ denote the number of items for color c in $\mathcal{B}(t)$. Let $n_c^O(t)$ be the number of items in the LP at time t for color c that have been processed by at most $1/2 + \epsilon$. Let $C_s(t)$ contain all colors c where $0 < n_c^A(t) \leq \frac{k}{\log^3 k\gamma}$ and $C_b(t)$ contain all colors c where $n_c^A(t) > \frac{k}{\log^3 k\gamma}$. Let $E^O(t)$ be the set of items that have been processed by at most $1/2 + 2\epsilon$ in the LP at time t that are not in $\mathcal{B}(t)$, i.e. $E^O(t) := \{i \notin \mathcal{B}(t) \mid i \leq t, \beta_{i,t} \leq 1/2 + 2\epsilon\}$. Let $c^*(t)$ be the color such that batches in the LP for color $c^*(t)$ that intersect time t is greater than 1/2, if it exists. Let $v_{c,t}^O = \sum_{i,c(i)=c} 1 - \beta_{i,t}$ denote the remaining volume of items for color c in the LP at time t.

Algorithm:

`Rule (i)`If there is an item in $i \in \mathcal{B}(t)$ processed by ϵ in the LP, switch to color $c(i)$.

`Rule (ii)`If there is an item $i \in \mathcal{B}(t)$ that is α ready at time t, switch to color $c(i)$.

`Rule (iii)`If there is a color c where $n_c^A(t) \geq k/10$, switch to color c.

`Rule (iv)`If the LP has processed items in $\mathcal{B}(t)$ corresponding to colors in $C_s(t)$ by a total of at least $\frac{|E^O(t)|}{8} + \frac{k}{2\log k\gamma}$ by time t then switch to the color of minimum weight that is not $c^*(t)$.

`Rule (v)`We perform this rule if none of the others apply. In this case, the algorithm switches to a color $c \in C_b(t)$ such that $n_c^A(t) \geq \frac{10}{11} v_{c,t}^O$. (We can show that such a color exists.)

References

1. Anna Adamaszek, Artur Czumaj, Matthias Englert, and Harald Räcke. Almost tight bounds for reordering buffer management. In *STOC*, pages 607–616, 2011.
2. Anna Adamaszek, Artur Czumaj, Matthias Englert, and Harald Räcke. Optimal online buffer scheduling for block devices. In *STOC*, pages 589–598, 2012.
3. Noa Avigdor-Elgrabli and Yuval Rabani. An improved competitive algorithm for reordering buffer management. In *SODA*, pages 13–21, 2010.
4. Noa Avigdor-Elgrabli and Yuval Rabani. An improved competitive algorithm for reordering buffer management. In *FOCS*, pages 1–10, 2013.
5. Noa Avigdor-Elgrabli and Yuval Rabani. An optimal randomized online algorithm for reordering buffer management. *CoRR*, 1303.3386, 2013.
6. Dan Blandford and Guy Blelloch. Index compression through document reordering. In *Proceedings of the Data Compression Conference*, DCC '02, pages 342–, Washington, DC, USA, 2002. IEEE Computer Society.
7. Niv Buchbinder and Joseph Naor. The design of competitive online algorithms via a primal-dual approach. *Foundations and Trends in Theoretical Computer Science*, 3(2-3):93–263, 2009.
8. Ho-Leung Chan, Nicole Megow, René Sitters, and Rob van Stee. A note on sorting buffers offline. *Theor. Comput. Sci.*, 423:11–18, 2012.
9. Matthias Englert, Harald Räcke, and Matthias Westermann. Reordering buffers for general metric spaces. *Theory of Computing*, 6(1):27–46, 2010.
10. Matthias Englert and Matthias Westermann. Reordering buffer management for non-uniform cost models. In *ICALP*, pages 627–638, 2005.
11. Iftah Gamzu and Danny Segev. Improved online algorithms for the sorting buffer problem on line metrics. *ACM Transactions on Algorithms*, 6(1), 2009.
12. K. Gutenschwager, S. Spiekermann, and S. Vos. A sequential ordering problem in automotive paint shops. *Intl J. of Production Research*, 42(9)(9):1865–1878, 2004.
13. Sungjin Im and Benjamin Moseley. New approximations for reordering buffer management. In *SODA*, pages 1093–1111, 2014.
14. Jens Krokowski, Harald Räcke, Christian Sohler, and Matthias Westermann. Reducing state changes with a pipeline buffer. In *VMV*, page 217, 2004.
15. Harald Räcke, Christian Sohler, and Matthias Westermann. Online scheduling for sorting buffers. In *ESA*, pages 820–832, 2002.