Cost-Oblivious Reallocation for Scheduling and Planning

Michael A. Bender
Stony Brook University
bender@cs.stonybrook.edu

Martín Farach-Colton
Rutgers University
farach@cs.rutgers.edu

Sándor P. Fekete
TU Braunschweig
s.fekete@tu-bs.de

Jeremy T. Fineman
Georgetown University
jfineman@cs.georgetown.edu

Seth Gilbert
NUS
seth.gilbert@comp.nus.edu.sg

ABSTRACT

In a reallocating-scheduler problem, jobs may be inserted and deleted from the system over time. Unlike in traditional online scheduling problems, where a job's placement is immutable, in reallocation problems the schedule may be adjusted, but at some cost. The goal is to maintain an approximately optimal schedule while also minimizing the ***reallocation cost*** for changing the schedule.

This paper gives a reallocating scheduler for the problem of assigning jobs to p (identical) servers so as to minimize the sum of completion times to within a constant factor of optimal, with an amortized reallocation cost for a length-w job of $O(f(w) \cdot \log^3 \log \Delta)$, where Δ is the length of the longest job and $f()$ is the reallocation-cost function. Our algorithm is ***cost oblivious***, meaning that the algorithm is not parameterized by $f()$, yet it achieves this bound for any subadditive $f()$. Whenever $f()$ is strongly subadditive, the reallocation cost becomes $O(f(w))$.

To realize a reallocating scheduler with low reallocation cost, we design a ***k-cursor sparse table***. This data structure stores a dynamic set of elements in an array, with insertions and deletions restricted to k "cursors" in the structure. The data structure achieves an amortized cost of $O(\log^3 k)$ for insertions and deletions, while also guaranteeing that any prefix of the array has constant density. Observe that this bound does not depend on n, the number of elements, and hence this data structure, restricted to $k \ll n$ cursors, beats the lower bound of $\Omega(\log^2 n)$ for general sparse tables.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Sequencing and scheduling*

Supported in part by NSF grants CNS 1408695, CCF 1439084, IIS 1247726, IIS 1251137, CCF 1217708, CCF 1114809, IIS 1247750, CCF 1114930, CCF 1218188, CCF 1314633, DFG Grant FE407/17-1, Tier 2 ARC MOE2014-T2-1-157, and Sandia National Laboratories.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SPAA'15, June 13–15, 2015, Portland, OR, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3588-1/15/06 ...$15.00.
DOI: http://dx.doi.org/10.1145/2755573.2755589 .

Keywords

Resource allocation; reallocation; cost-oblivious problems

1. INTRODUCTION

Scheduling is a ubiquitous aspect of life, and an essential aspect of real scheduling is that *schedules change*. Consider train and plain schedules: both are examples of schedules prone to disruption, where weather patterns and mechanical failures affect the availability of crew and aircraft.

In many such cases, there is a cost when resource allocations change (e.g., people miss flights, airlines have to pay overtime, etc.), so the objective is to minimize the cost of schedule disruptions (or resource reallocation) without violating scheduling constraints.

Scheduling subject to changes/disruptions is a classic example of a reallocation problem. In a reallocation problem, an allocator assigns resources to jobs. Unlike in tradition online problems, the allocator may *revise prior decisions, at some cost*. A good reallocation algorithm maintains an approximately optimal allocation of resources, while simultaneously minimizing the reallocation cost. Reallocation is online in the sense that the schedule is modified immediately when a new request arrives, however no decision is irrevocable and resources may always be reallocated.

Perhaps the most-studied problem with a reallocation flavor is multi-machine load-balancing (e.g., [1, 6, 16, 28, 30, 33, 34]), where a solution should simultaneously minimize the maximum machine load and the number of migrations across machines. When load-balancing a web server, for example, it makes sense to optimize for migration cost. In other scheduling problems, such airline scheduling, changing the start time of a job may be expensive and the goal is to minimize schedule updates.

In our experience, we have seen reallocation problems underlying many aspects of parallel systems. In our own research, we have modelled FPGA reorganization as a reallocation problem, minimizing reconfiguration costs [17]. We have also explored reallocation problems in database resource management with respect to crash safety and transactional support [8]. One of our long-term goals is to understand which scheduling objectives are amenable to efficient reallocation, and which ones are brittle.

Reallocation and Scheduling

This paper addresses the problem of minimizing the sum of completion times with reallocation in a system of p (identical) servers. The input consists of an online sequence of job

insertions or deletions, where each job has a size w_j and a (re)scheduling cost $f(w_j)$ that depends only on its size.

The goal is to maintain, after each insert/delete request, a schedule that (approximately) minimizes the sum of completion times of all jobs currently in the system. We abbreviate this problem as $p|f(w)\ realloc\ |\sum_j C_j$ (generalizing standard notation [18]).

The optimal static solution for sum of completion times is achieved by sorting jobs by increasing length and assigning them greedily to servers, allocating each job in turn to the first available server. However, it is not hard to show that achieving this optimal solution could require a large number of reallocations after each insert/delete.

To achieve low reallocation cost, we allow solutions that approximately minimize the sum of completion times. A natural way to *approximate* the optimal sum of completion times is to sort jobs by approximate size, e.g., schedule all jobs with size at most 2^i before larger jobs with size at most 2^{i+1}. Our solution adopts this approximate-sorting approach, but the difficulty is in maintaining this order with low reallocation cost as insertions and deletions occur.

There is a special case worth noting. When $f(w) = 1$ for all w, i.e., moving a job has constant cost regardless of size, there is a straightforward algorithm with $O(1)$ amortized reallocation cost that maintains a sum of completion times no more than 4 times the optimal.[1]

A goal of this paper is to explore reallocating schedulers for general reallocation-cost functions f, not just constant f. If $f(w) = w$, i.e., the cost is linear in the size of a job, the simple solution degrades, giving a reallocation cost of $O(\log \Delta)$, where Δ is the size of the largest job.

One option is to design a suite of algorithms, each specifically tuned to achieve a better reallocation cost for a specific function f. However, for some functions, e.g., the linear cost function $f(w) = w$, we know of no better solution than the one presented in this paper.

Our more elegant alternative to optimizing for specific cost functions is to produce a single algorithm that simultaneously achieves the best known reallocation cost for a wide range of reallocation-cost functions.

Our main result is a ***cost oblivious*** reallocating scheduler that has low reallocation cost for any subadditive, monotonically nondecreasing cost function f. (A monotonically nondecreasing function $f(x)$ is ***subadditive*** if $f(x + y) \leq f(x) + f(y)$ for any positive x and y. All monotonic concave functions are subadditive.) Being cost oblivious means that the scheduler does not use any knowledge of f. Cost obliviousness was introduced in [8], motivated by the practical problem of storage reallocation in database systems. Cost obliviousness is desirable, because the true cost of reallocation may be complex or even unknown.

Our algorithm achieves an $O(1)$ reallocation cost for constant f and an $O(\log^3 \log \Delta)$ reallocation cost for linear f.

[1]Consider the single-server case, where $p = 1$. Allocate a job-sized gap in the schedule between each group of jobs. To insert a size-2^i job, schedule it immediately after the last size 2^i job, possibly evicting an overlapping larger job which is rescheduled recursively. This process may cascade across all sizes, but it is not hard to show that the amortized reallocation cost is $O(1)$ if $f(w) = 1$ since large-job evictions leave large gaps that facilitate many future small-job insertions.

Formalization of the reallocating scheduler problem

An ***online execution*** consists of a sequence of requests of the form $\langle$InsertJob, *name*, *length*$\rangle$ and $\langle$DeleteJob, *name*$\rangle$, with integral lengths. Between requests, we define the ***active jobs*** to be those that have been inserted but not yet deleted. We typically assume throughout that we are given in advance Δ, the size of the largest job; we discuss how to remove this restriction. There is no given bound on the total number of jobs or the total number of active jobs.

After each insert/delete request, the scheduler must output a schedule for each server. If $\mathcal{S}$ and $\mathcal{S}'$ are schedules before and after a request r, then the ***reallocation cost of r*** is the sum of the reallocation costs of all jobs whose scheduling has changed between $\mathcal{S}$ and $\mathcal{S}'$. The reallocation cost of a sequence of requests is the sum of the reallocation costs for each request.

There are two different types of reallocations that may occur. If a job's schedule is modified, but it remains scheduled on the same server, it is considered a ***nonmigrating*** reallocation. If the job is rescheduled on a new server, it is considered a ***migrating*** reallocation.

A reallocating scheduler A is ***(f, a, b)-competitive*** for cost function $f()$, if (1) the objective function is always within an a-factor of optimal, and (2) the reallocation cost is at most b times the sum of the allocation costs of every object inserted thus far (including those that have subsequently been deleted).

Let $\mathcal{C}$ be a set of cost functions. A reallocation algorithm A is ***cost oblivious*** if it does not depend on $f()$. (This means not only that $f()$ is not a parameter to algorithm A, but also A learns nothing about $f()$ as A executes.) A cost-oblivious reallocator A is ***$(\mathcal{C}, a, b)$-competitive*** if it is (f, a, b)-competitive for every $f \in \mathcal{C}$; we abbreviate to ***(a, b)-competitive*** if the set $\mathcal{C}$ is unambiguous.

Results

We develop a reallocating scheduler for p servers that achieves an $O(1)$ approximation for the sum of completion times. For subadditive cost functions, the scheduling algorithm is

$$\big(O(1), O(\log^3 \log \Delta)\big)$$

competitive, where Δ denotes the length of the longest job. For strongly subadditive cost functions, the algorithm is $(O(1), O(1))$-competitive. (We define a subadditive function $f(x)$ to be ***strongly subadditive*** if $f(2x) \leq (2-\gamma)f(x)$ where $2 > \gamma > 0$ is bounded above 0 by a constant.) This scheduling algorithm is cost oblivious, and the same algorithm achieves both bounds. All the reallocations associated with insertions are nonmigrating, and each deletion triggers at most one migrating reallocation.

The key technical tool is a single-server scheduler that achieves a $1 + \varepsilon$ approximation ($0 < \varepsilon \leq 1/2$) for the sum of completion times. We then build the general multi-server scheduler by properly load balancing jobs across servers.

To achieve these results, we develop a ***k-cursor sparse table*** that maintains constant prefix density in an array with inserts/deletes restricted to k "cursors" in the array. This data structure has an amortized $O(\log^3 k)$ insertion and deletion cost, which (critically) is independent of the number n of elements. We employ the structure with $k = \Theta(\log \Delta)$, giving the $O(\log^3 \log \Delta)$ bound.

In contrast, general sparse tables have an amortized update cost of $\Theta(\log^2 n)$ [21, 35–37], which is tight [11]. Re-

placing the k-cursor sparse table with a general sparse table in the scheduling algorithm of Section 2 would yield a significantly worse reallocation cost of $O(\log^3 V)$, where $V \geq \Delta$ is the total length of all jobs. Our k-cursor structure has significant complexity beyond that of a standard sparse table, with many more structural constraints to guarantee the stronger bound for the special case of cursors.

Other related work

Reallocation-style problems have been studied in a variety of contexts, particularly in the last several years. In the context of scheduling, we previously studied a reallocation scheduler for unit-sized jobs with arrival times and deadlines [7]. Some aspects of that problem were much simpler: all jobs had an identical size of one. Some aspects were more complicated: jobs could not be scheduled arbitrarily, but had to respect arrival times and deadlines. The techniques from [7] do not apply to the problem addressed in the current paper.

Cost oblivious reallocation is a relatively new concept that we recently introduced [8]. In that paper, the goal was to minimize the total space footprint of the data being stored; this is analogous to minimizing the makespan (i.e., the maximum time that any job completes) in scheduling. In the current paper, we focus on minimizing the sum of completion times, which imposes different constraints on the ordering of jobs. The storage reallocator [8] relied on different tools, e.g., cascading buffers, that would not help for minimizing sum of completion times; by contrast, the main tool here is the k-cursor sparse table.

Shachnai et al. [29] explore a form of reallocation for combinatorial optimization. Given an input, an optimal solution for that input, and a modified version of the input, they develop algorithms that find the minimum-cost modification of the optimal solution to the modified input. A difference between their setting and ours is that we measure the ratio of reallocation cost to allocation cost, whereas they measure the ratio of the actual transition cost to the optimal transition cost resulting in a good solution. Also, we focus on a sequence of changes, meaning we amortize the expensive changes against a sequence of updates.

Davis et al. [14] study a reallocation problem, where an allocator divides resources among a set of users, updating the allocation as the users' constraints change. The goal is to minimize the number of changes to the allocation. Other papers that solve specific instances of reallocation problems include [15, 17, 19, 28, 32].

Robust scheduling is a related notion, which involves designing schedules that can tolerate some uncertainty [12, 13, 22, 25–27, 31]. The assumption is that the problem is approximately static, but there is some error or uncertainty. The schedule remains near optimal even if the underlying situation changes.

Reoptimization problems minimize the computational cost for incrementally updating the schedule [2–5, 10]. By contrast, we focus on the resource-allocation cost instead of the computation cost of finding an allocation.

One of the contributions of this paper is a new sparse table data structure; many sparse tables have appeared previously in the literature [9, 11, 20, 21, 24, 35–37]. Indeed, one distinction between our paper and most prior work on reallocation and rescheduling is that the technical results seem to meld data structures and combinatorial optimization.

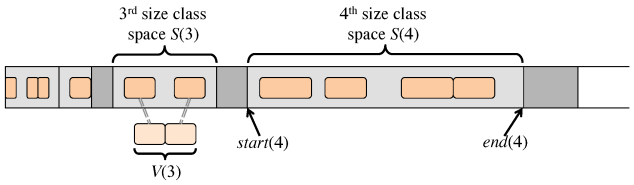

Figure 1: The layout of the schedule, viewed as an array, with $\delta = 0.5$. The light-gray rectangles are the regions assigned to each size class. The orange rounded rectangles are the jobs scheduled in each size class. The dark-gray rectangles are regions outside size classes and contain no jobs. Note that the 4th size class contains jobs of slightly different sizes in arbitrary order.

2. SINGLE SERVER SCHEDULING

This section presents a cost-oblivious reallocating scheduler for the problem of (approximately) minimizing the sum of completion times for the single-server case where $p = 1$. The algorithm achieves an efficient reallocation cost on job insertions and deletions, as stated by the following theorem, proved at the end of the section.

THEOREM 1. *For any constant ε with $0 < \varepsilon \leq 1$, there exists a cost-oblivious reallocating scheduler for $p = 1$ for minimizing the sum of completion times that is $(1 + \varepsilon, O((1/\varepsilon^5)\log^3 \log_{1+\varepsilon} \Delta))$-competitive over all subadditive cost functions, and $(1 + \varepsilon, O(1/\varepsilon^3))$-competitive over all strongly subadditive cost functions.*

On a single server, the optimal schedule for minimizing sum of completion times is to sort the jobs in order of increasing size; see [23]. The main goal of our algorithm is to maintain this order approximately by allowing jobs of roughly the same size to be in arbitrary order. Our algorithm organizes jobs into groups called ***size classes***, containing jobs that differ in size by no more than a $1+\delta$ factor, for constant $\delta \leq 1$ with $\delta = \Theta(\varepsilon)$. (We shall specify δ more precisely in the analysis.) Precisely, a job of size w belongs to size class $j = \left\lfloor \log_{(1+\delta)}(w) \right\rfloor$, i.e., it belongs to size-class j if $(1+\delta)^{j-1} \leq w < (1+\delta)^j$.

Schedule-array layout

Think of the jobs as being placed in an array, as shown in Figure 1. The schedule array is divided into $\left\lceil \log_{(1+\delta)} \Delta \right\rceil$ segments, where the ith segment contains jobs from size class i. Let $V(i)$ denote the ***volume***, or total length, of jobs in size class i, and let $S(i)$ denote the total space allocated in the array for size class i (possibly including empty space). We write $V(i,j) = \sum_{\ell=i}^{j} V(\ell)$ for the sum of the volumes of size classes i to j. Moreover, $\mathit{start}(i)$ and $\mathit{end}(i)$ are, respectively, the beginning and end of size class i.

The key challenge is managing the size-class boundaries in the array: each size class needs enough space to accommodate insertions, but sufficiently little to ensure that the sum of completions times is approximately optimal. The following property limits the amount of space:

PROPERTY 1. *For every $j \leq \log_{(1+\delta)} \Delta$:*

- $S(j) \geq \lfloor V(j)(1+\delta) \rfloor$,
- $\mathit{start}(j) \leq V(1, j-1)(1+\delta)^2$, *and*

- $end(j) \leq V(1,j)(1+\delta)^2$.

That is, for every size class j, there is at least a factor of $(1+\delta)$ of extra allocated space (modulo rounding), and at most a factor of $(1+\delta)^2$ of extra allocated space. Note there may be additional empty space left between size classes (to facilitate future growth and shrinking).

As insertions and deletions occur, the boundaries of size classes may change, but these movements must be limited to avoid expensive reallocations. Job movements within each size class must also be bounded, but these are easier to handle. We use the term ***lost slots*** to refer to any array slots that were part of a particular size class before an operation, but that are not part of that size class after. A second key property is that the algorithm supports the following "one directional" charging scheme for lost slots:

PROPERTY 2. *There exists a function g mapping lost slots to insert/delete operations subject to:*

- *The function g maps each slot lost by size class j to an insert/delete operation on a size class $\leq j$.*
- *There are $O(\log^3 \log_{1+\delta} \Delta/\delta^3)$ lost slots in total mapped to any particular insert/delete operation.*
- *There are $O(1)$ lost slots from each size class mapped to any particular insert/delete operation.*

To manage size-class boundaries dynamically, we employ a new data structure called a ***k-cursor sparse table***. A k-cursor sparse table is a special case of a sparse table that contains k different regions called ***cursor districts*** (corresponding to size classes), each of which supports insertions and deletions of unit-size elements with amortized cost $O(\log^3 k)$. The k-cursor data structure has constant "prefix density," which implies Property 1, and it has "one directional rebalances" consistent with Property 2. Perhaps counterintuitively, we do not use the k-cursor table to reallocate jobs directly, but use it to indicate when size-class boundaries move and contained jobs should be reallocated.

We now describe the reallocation algorithm, ending with a proof of Theorem 1. The k-cursor data structure, hiding much of the technical complexity, is described in Section 4.

Reallocating scheduler algorithm

We now describe the basic algorithm. Every job is stored in array A, which is divided into $\left\lceil \log_{(1+\delta)} \Delta \right\rceil$ size-classes.[2] We use an auxiliary k-cursor data structure, where $k = \left\lceil \log_{(1+\delta)} \Delta \right\rceil$, to maintain the boundaries of job classes; each district boundary is interpreted as a size-class boundary.

Insertions: Consider a job of size w belonging to size class $j = \left\lfloor \log_{(1+\delta)}(w) \right\rfloor$. (Recall that w is an integer between 1 and Δ.) Let $\tilde{w}$ be the minimum size of a job in size class j. In order to insert the job:

1. Insert (roughly) $w(1+\delta)$ elements into district j of the k-cursor structure, more specifically, if $V(j)$ is the current volume of size class j, including the new job, insert until the number of elements in the district equals $\lfloor V(j)(1+\delta) \rfloor$.
2. Adjust the size classes in the array A to match the district boundaries indicated by the k-cursor structure. No jobs have moved yet.
3. Identify S, the set of jobs overlapping lost slots, i.e., those falling before the new starting point of their size class's segment. These jobs must be moved.
4. For each job in S from largest to smallest size class, remove it from its old location and re-***place*** it in its size class, as described below.
5. Finally, place the newly inserted job in its size class.

To ***(re)place*** a job of length w in size class j, we identify a subinterval in the job class that has at least w empty space and is of size at most $O(w/\delta)$ (i.e., it contains at most $O(1/\delta)$ jobs). We then rearrange the jobs in this subinterval to make space, and (re)place the job.

One further restriction is that this subinterval should not include the first or last $\lfloor \tilde{w}\delta/4 \rfloor$ slots of the size class so as to avoid further changes should the size class boundary move again. (Recall that $\tilde{w}$ is the minimum size job for the size class.) We refer to the first and last $\lfloor \tilde{w}\delta/4 \rfloor$ slots as ***boundary padding***, and the remaining slots as ***non-boundary slots***. Whenever boundary movement causes jobs to shift, no job should be replaced within the first or last $\lfloor \tilde{w}\delta/4 \rfloor$ slots of the size class. This boundary padding prevents small changes in the boundary from causing jobs to move.

We now explain in more detail how a job is (re)placed. We begin with the case where $V(j) < 2/\delta$. Since there is at least one job in the size class, we know that $\tilde{w} \leq V(j) < 2/\delta$. Hence the boundary padding is of size $\lfloor \tilde{w}\delta/4 \rfloor = 0$. Thus, we simply rearrange all $< 2/\delta$ jobs to make room.

Next, we consider the case where $2/\delta < V(j) \leq 5w/\delta$. In this case, we have $\lfloor V(j)(1+\delta) \rfloor \geq V(j)(1+\delta/2)$. Consider the entire size class, excluding the boundary padding of size $\lfloor \tilde{w}\delta/4 \rfloor$. Since $V(j) \geq \tilde{w}$, we conclude that the excluded boundary padding contain at most $V(j)\delta/2$ slots, and hence there are at least $V(j)$ available non-boundary slots. Again, we move every job in the non-boundary portion of the size class to make room for the (re)placed one, moving at most $10/\delta$ jobs (as each job is of size at least $w/(1+\delta)$).

Lastly, assume $V(j) > 5w/\delta$. We identify a subinterval in size class j of length at most $10w/\delta$ that has at least w empty space and does not overlap the boundary padding.

First, we determine the percentage of empty slots in the non-boundary portion of the size class. Since (in this case) $V(j) > 5w/\delta \geq 4/\delta$, the total number of slots is at least $\lfloor V(j)(1+\delta) \rfloor \geq V(j)(1+3\delta/4)$. At most $\tilde{w}\delta/2 \leq V(j)\delta/2$ of these slots are boundary slots, and so the total number of non-boundary slots is at least $V(j)(1+\delta/4)$. Therefore, the fraction of free non-boundary slots in the size class is at least $1 - V(j)/V(j)(1+\delta/4) \geq \delta/(4+\delta)$.

Next, partition the non-boundary portion of the size class into (disjoint) subintervals of length between $5w/\delta$ and $10w/\delta$. (The total number of slots in the class is at least $5w/\delta$ in this case.) At least one of these intervals must have at least the average number of empty slots, i.e., for at least one interval, the fraction of free slots is at least $\delta/(4+\delta)$.

In this subinterval, the amount of free space is at least $(5w/\delta)(\delta/(4+\delta)) \geq w$. To make room, move all the jobs in the subinterval; there are at most $10/\delta$ jobs. We conclude:

CLAIM 2. *Given fixed size-class boundaries obeying Property 1, each job (re)placement causes at most $O(1/\delta)$ jobs within the same size class to move.* □

[2]Due to interactions with the k-cursor structure, we assume here that Δ, the size of the largest job, is known in advance. This assumption can be removed as the k-cursor data structure allows addition and removal of the last district.

Deletions: Deletions differ only slightly from insertions. First, the size-w job is removed from the array. Then, roughly $(1+\delta)w$ elements are removed from the k-cursor structure. Finally, adjust size-class boundaries and replace jobs as before, proceeding from smallest to largest size class.

Analysis

We now analyze the amortized reallocation cost of our algorithm. Combining the following lemma with a setting of $\delta = \Theta(\varepsilon)$ proves the reallocation-cost aspect of Theorem 1.

LEMMA 3. *Consider an execution of the reallocation scheduler. If the cost function f is subadditive, then the amortized reallocation cost per operation is at most $O\left((1/\delta^5)\log^3 \log_{1+\delta}\Delta\right)$ times the initial allocation cost. If f is strongly subadditive, then the amortized reallocation cost is at most $O(1/\delta^3)$ times the initial allocation.*

PROOF. Consider each job of size w as consisting of w unit-sized components. Our goal is to map the reallocation of each unit-sized component to some unit-sized component of an insertion or deletion in an earlier job class. More precisely, for an insertion/deletion of a size-w job, we will show that at most $O(w\log^3 k/\delta^5)$ unit-size components, all in the same or later size classes, are charged to the operation. The last step of the proof is to substitute $k = \log_{1+\delta}\Delta$.

Such a mapping implies the desired result for subadditive cost functions due to subadditivity and monotonicity. Subadditivity implies that the per-unit cost of reallocating larger jobs is less than that of smaller jobs. Monotonicity implies that all jobs in the same size class have a per-unit cost that varies by at most a constant factor.

Due to the boundary padding, there must be at least $\Omega(\lceil\delta w\rceil) = \Omega(\delta w)$ unit boundary movements to effect the reinsertion of a job in a size class having jobs of size at least w. (The ceiling arises because at least one slot must be lost to cause any job to move.) Due to Claim 2, the cost of this reinsertion is at most $O(1/\delta)$ times the initial allocation, or $O(1/\delta^2)$ per unit boundary slot lost.

Multiplying the lost slots of Property 2 with the $O(1/\delta^2)$ movements caused by each lost slot proves the lemma for subadditive functions. For strongly subadditive functions, observe that the per-unit cost of reallocating jobs geometrically decreases every $\log_{1+\delta} 2 = 1/\lg(1+\delta) \approx 1/\delta$ size classes. Applying the other bound ($O(1)$ lost slots per district or size class) of Property 2 and multiplying by this additional $1/\delta$ factor completes the proof. □

We now bound the sum of completion times to complete the proof of Theorem 1. The analysis actually shows a $(1+O(\delta))$ ratio; here is where we choose the constant in $\delta = \Theta(\varepsilon)$. The proof follows from the fact that the jobs are stored in order, and the empty space is limited by Property 1.

LEMMA 4. *At all times, the scheduled sum of completion times is at most $(1+\varepsilon)$ times larger than optimal.*

PROOF. Fix a job class j containing jobs of size at least s and at most $s(1+\delta)$. We show that the sum of completion times of the jobs in class j are within a $1+O(\delta)$ factor of the sum of completion times for these same jobs in the optimal schedule. Let J be the set of jobs in job class j, and let $k = |J|$ be the number of jobs in J.

We divide the analysis into two parts, first examining the contribution of jobs in size classes $< j$, and then in size class j, to the sum of completion times of jobs in class j.

Recall that in an optimal schedule, the jobs are sorted in order from smallest to largest. Thus, every job that precedes job class j in the schedule also precedes every job in J in the optimal schedule. For each job in J, these earlier jobs contribute $V(1,j-1)$ to the sum of completion times in the optimal schedule, i.e., at least $kV(1,j-1)$ in total.

By Property 1, guaranteed by the k-cursor structure, job class j begins no later than $V(1,j-1)(1+\delta)^2$, and hence in the actual schedule these jobs contribute at most $kV(1,j-1)(1+\delta)^2$. The contribution of the jobs prior to job class j to the sum of completion times is within a $1+3\delta$ factor.

We now focus on the contribution of the jobs and empty space in job class j to the sum of completion times. Since there are k jobs of size at least s, the contribution of these jobs is at least $OPT_j = k(k+1)s/2 \geq sk^2/2$.

We now consider the delay caused by the empty space that is part of job class j. Recall that job class j has size at most $V(j)(1+\delta)^2$, and hence there is at most $3\delta V(j)$ empty space. Also, notice that since every job is of size at most $s(1+\delta)$, we know that $V(j) \leq ks(1+\delta) \leq 2ks$. Thus, the empty space contributes at most $k3\delta V(j) \leq 6\delta k^2 s$ to the sum of completion times, i.e., at most $12\delta OPT_j$.

Finally, we consider the delay caused by the jobs being out of order within job class j. Imagine beginning with the jobs in J in sorted order, and swapping jobs one at a time until the schedule matches that of the actual execution. This can be accomplished using at most k swaps, starting with the first job in the schedule and proceeding onwards. Since all the jobs have size at least s and at most $s(1+\delta)$, each swap delays each later job by at most δs. This reordering adds $\delta k^2 s \leq 2\delta OPT_j$ to the sum of completion times.

In total, if OPT is the optimal schedule, then the schedule has a sum of completion times of at most $OPT(1+17\delta)$. □

3. PARALLEL SCHEDULING

This section generalizes the result from the previous section, showing how to design a reallocating scheduler for a parallel system consisting of p servers. The main result of this section is a scheduler where the sum of completion times is an $O(1)$ approximation of optimal, while the cost of insertions/deletions is $O(\log^3 \log \Delta)$ competitive with the total allocation costs (both bounds independent of p).

Algorithm

We consider a straightforward application of the previous scheduler. Each server executes an independent instance of the 1-processor reallocating scheduler.

As jobs are inserted and deleted, we use a simple balancing rule to ensure that within each size class, jobs are distributed evenly among the servers.

INVARIANT 5. *For any two servers s_k and s_ℓ, for every size class j, the number of jobs in size class j on the two servers differs by at most* 1.

In more detail, when an insertion happens: If the new job is in size class j, then we insert it on the server that has the smallest number of jobs in size class j (breaking ties by server id). That is, for each size class, we insert jobs in round-robin order. As such, there are no job migrations during an insertion. Thus each insertion has cost $O(\log^3 \log \Delta)$, i.e., the cost of scheduling one job on one server.

When a deletion happens, we may need to migrate a single job in order to maintain balance. Assume the job being

deleted is in size class j and is deleted from server s_ℓ. If, after the deletion, Invariant 5 still holds, then we are done. Otherwise, we need to migrate one job to restore the invariant. Again, the cost is $O(\log^3 \log(\Delta)$: two deletions and one insertions yielding one migration.

Analysis

We show that this algorithm ensures that the sum of completion times is within a constant factor of optimal. For a given set of scheduled jobs, we consider the modified collection of jobs where the size of each job is rounded up to the nearest power of two. The real sum of completion times is bounded by the sum of completion times of the rounded set. We first observe the standard fact (see, e.g., [23]):

LEMMA 6. *Given a set of jobs in sorted order of size from smallest to largest, the sum of completion times is minimized by assigning jobs to servers round-robin.*

We next compare the schedule constructed by the optimal round-robin algorithm and the schedule constructed by our algorithm. The key difference is that when we start a new job class, we restart the round-robin scheduling with the first server. That is, within a job class, jobs are assigned round-robin across servers (in the order that they were added); but across job classes this is not true. We observe the following:

LEMMA 7. *For each job j, let S_j be the set of jobs that precede it on its server, as scheduled by the reallocating scheduler, and let S_j^{OPT} be the set of jobs that precede it on its server, as scheduled by the optimal round-robin schedule. The set difference $S_j \setminus S_j^{OPT}$ contains at most one job per size class.*

This immediately yields the following corollary:

COROLLARY 8. *The difference between the completion time of a job j when scheduled according to the reallocating scheduler and the optimal round-robin scheduler is at most $2size(j)$.*

Thus, the sum of completion times increases by at most a factor of two. This proves the following theorem:

THEOREM 9. *For a system with p servers, there exists a cost-oblivious reallocating scheduler for minimizing the sum of completion times that is $(O(1), O(\log^3 \log \Delta))$-competitive over all subadditive cost functions, and $(O(1), O(1/\varepsilon^3))$-competitive over all strongly subadditive cost functions. An insertion incurs only nonmigrating reallocations, and a deletion incurs at most one migrating reallocation.*

4. THE K-CURSOR DATA STRUCTURE

This section presents the ***k-cursor sparse table***. A k-cursor table consists of k disjoint regions, called ***cursor districts***, and supports insertion and deletion of the n unit-sized elements into a specified district. At any time, the districts must be stored in order in an (infinite) array, subject to ***constant prefix density***. Specifically, given space parameter $\delta > 0$, the first x elements (i.e., those in the earliest districts) must be stored within the earliest $x + \lfloor \delta x \rfloor$ array slots.[3]

[3]For purposes of exposition, we allow $O(1)$ additional information per district to be stored outside the array, e.g., keeping pointers to the district's boundaries. In the scheduling reallocation problem, arbitrary bookkeeping is permitted.

Each cursor district acts as a LIFO stack, where elements are inserted at the end of each district, and must be stored in each district in order of insertion. Similarly, elements must be deleted in reverse order of insertion. This is why we think of the data structure as consisting of *cursors*, e.g., editing at k locations within a single file. More precisely, the k-cursor sparse table supports:

- INSERT(x, j)—Insert the new element x at the end of the jth cursor district.
- DELETE(j)—Delete and return the last element from the jth cursor district, if the district is not empty.

Assume k is known *a priori*; we later relax this assumption.

Like standard sparse-tables [21], where elements may be inserted or deleted at arbitrary positions, we leave "empty space" in the array to support operations with a lower amortized cost. On insertion or deletion, an occasional "rebalance" may be triggered, where a (potentially large) region of the array is rebuilt, making room for future insertions or compacting the storage on deletion. Standard sparse tables are more general, but the amortized cost for insertion and deletion is $O(\log^2 n)$, which has a matching lower bound [11] in the worst case.

Our k-cursor sparse table supports insertions and deletions in amortized time $O(\log^3 k)$ per operation, with a constant that depends on the space parameter δ hidden in the big-O. Our k-cursor sparse table is also ***one directional***: an insert or delete triggers a rebalance region that extends only to the right (for Property 2 of Section 2).

The remainder of this section is organized as follows. Section 4.1 provides an overview of the data structure, which includes most of the important ideas and intuition, with some details omitted. Section 4.2 fills-in the remaining details of the algorithm. Section 4.3 gives the space analysis and sketches the performance analysis, with more complete proofs deferred to the full version of the paper.

4.1 Overview

At a high level, the districts are stored in order in an array, with empty space carefully distributed to facilitate faster insertions and deletions. How this space is arranged and redistributed/rebalanced on insertions and deletions is the key to an efficient structure. The empty spaces in our data structure take two forms, which we call "buffers" and "gaps." The gaps add an additional complication, but they only arise in the data structure when districts have drastically different sizes. We describe a gapless version first, and later augment it to include gaps in Section 4.2.

We group cursor districts at $H + 1$ levels of granularity ($H = \lceil \lg k \rceil$) as follows. Without loss of generality, assume k is a power of 2, and make a complete binary tree of height $H = \lg k$ where the cursor districts are the leaves of the tree. A ***level-0 chunk*** corresponds to a single district and its buffer, stored sequentially in the array. A ***level-$(i+1)$ chunk***, for $0 \le i < H$, corresponds to a height-$(i+1)$ subtree of districts, and thus includes the corresponding 2^{i+1} consecutive districts, as well as lower-level buffers and gaps. More precisely, a level-$(i+1)$ chunk contains two level-i chunks plus a level-$(i+1)$ buffer.[4] In the array, we

[4]The binary tree can be represented implicitly because the chunks are "aligned," meaning that the 0th level-i chunk corresponds to districts $0, 1, \ldots, 2^i - 1$, and in general the rth level-i chunk consists of districts $r2^i, r2^i + 1, \ldots, (r+1)2^i - 1$.

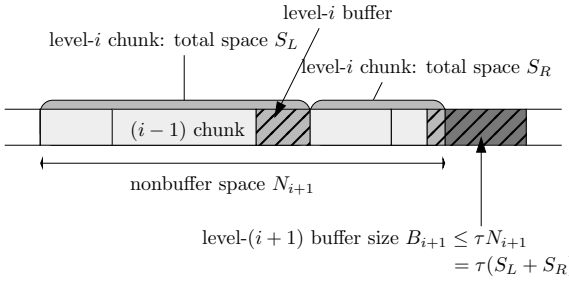

Figure 2: The recursive layout of a single level-$(i+1)$ chunk in the k-cursor sparse array, consisting of two level-i chunks and a level-$(i+1)$ buffer. An additional level of recursive layout is displayed.

store level-$(i+1)$ chunks recursively as its two child chunks, first the ***left chunk*** then the ***right chunk***, followed by a ***level-$(i+1)$ buffer***, as shown in Figure 2. Buffer slots are empty array slots that may be redistributed when insertions or deletions cause "rebalances" to occur. Every empty array slot (buffer and gap) is assigned to a particular chunk.

As a convention throughout this section, when referring to one chunk c_i we use the subscript i to denote its level. When referring to the left and right child chunks of a chunk, we use c_L and c_R, respectively.

The τ parameter: The data structure is parameterized by τ, which is set to $\tau = \Theta(\delta/H)$ for analysis. Since δ bounds the extra space, large δ only makes the problem easier; without loss of generality we assume $\delta \le 1$ and thus $\tau \le 1$. Due to rounding to integer indices within the array, it is convenient to restrict τ further. Specifically, we will choose τ such that $1/\tau \ge H$ is an integer.

Notation for space usage

For level-i chunk c_i, define space usage in the array as follows:

- ***Buffer space***, denoted by $B(c_i)$, is the number of (empty) array slots assigned to c_i's buffer.
- ***Gap space***, denoted by $G(c_i)$, is the number of (empty) array slots assigned to c_i as level-i gaps (see Section 4.2). A level-0 chunk has no gaps. For now, the reader should assume $G(c_i) = 0$.
- ***Total space***, denoted by $S(c_i)$, includes all array slots assigned to c_i or its nested children. For $i \ge 1$, if c_i has children c_L and c_R, then $S(c_i) = S(c_L) + S(c_R) + B(c_i) + G(c_i)$. For a level-0 chunk, its total space is its buffer space plus the number of elements in the corresponding district.
- ***Nonbuffer space***, denoted by $N(c_i)$, is defined as the total space excluding the buffer at that node but including its gaps, i.e., $N(c_i) + B(c_i) = S(c_i)$.

We maintain the following invariant over buffer/gap space, bounding the empty space with respect the total space of the children. This invariant alone is enough to bound the total space of the data structure, but it does not immediately imply prefix density, as we have not yet specified where the gaps are located.

INVARIANT 10. *(Space invariant) Except while processing an operation, for any chunk c_i we have:*

$$0 \le B(c_i) \le \tau N(c_i) \ .$$

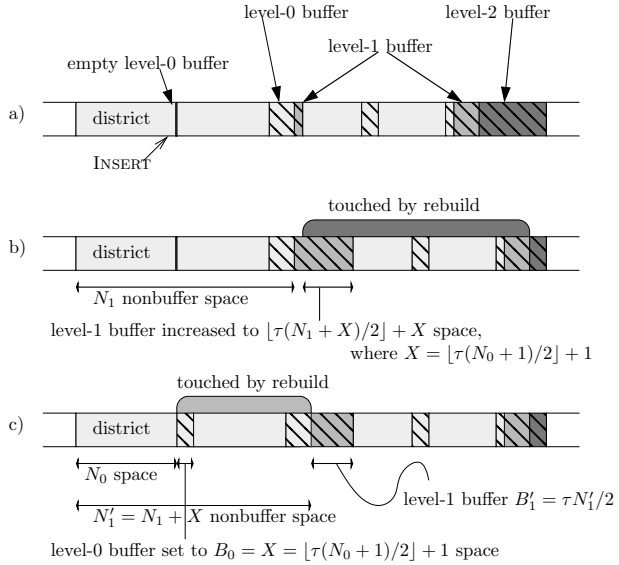

Figure 3: Example insertion into an empty buffer. (a) the initial layout of the structure. (b) the state after the level-1 buffer is rebuilt. (c) the state after the level-0 buffer is rebuilt. Finally (not shown), the element would be inserted into the newly rebuilt level-0 buffer, increasing and decreasing N_0 and B_0 by 1, respectively.

If c_i is a level-0 chunk, then $G(c_i) = 0$. Otherwise, let c_R be its right child, and we have:

$$0 \le G(c_i) \le \tau S(c_R) \ .$$

Insertion sketch

At a high level, ignoring gaps, insertions operate as described below. (See also Figure 3.) Section 4.2 gives more detail and pseudocode. To insert into a district, replace the leftmost buffer slot with the element being inserted. If no empty buffer slots remain, then the corresponding chunk must be "rebuilt." During a rebuild, the chunk increases its buffer space by taking slots from its parent chunk's buffer. "Taking slots" means scanning all slots between the current buffer and the parent buffer and sliding those elements to the right. If the parent does not have a large enough buffer to rebuild the child, the parent must first be rebuilt; this rebuild process may cascade up ancestors. When rebuilding a chunk c_i, it is rebuilt so that the *a posteriori* buffer size is $B'(c_i) = \lfloor \tau N'(c_i)/2 \rfloor$, where primed terms denote sizes after completion.

Analysis overview and why gaps occur

A main idea of the analysis (see Section 4.2) is that when rebuilding the buffer of a left level-i chunk c_L (the expensive case), the cost of the rebuild is proportional to the total space $S(c_R)$ of the right level-i chunk c_R that is moved out of the way. We amortize this cost against the $\ge \lfloor \tau N(c_L)/2 \rfloor$ slots moving into c_L's buffer. As long as the right level-i chunk satisfies $S(c_R) = O(N(c_L))$ slots, i.e., the right chunk is not much larger than the left chunk, then the amortized cost per buffer slot moved is $O(1/\tau)$. Summing across the

$H+1$ levels yields an amortized insertion cost of $O(H/\tau)$, which is $O(\log^2 k/\delta)$ for $H = \lceil \lg k \rceil$ and $\tau = \Theta(\delta/H)$.

This argument fails if the right chunk is much larger than the left chunk. Thus, we introduce "gaps." As with buffers, gaps are tagged with a level. The purpose of level-$(i+1)$ gaps is similar to level-$(i+1)$ buffers—when rebuilding a level-i chunk, take the nearest level-$(i+1)$ empty slots, either from the corresponding level-$(i+1)$ gaps or buffer. Careful gap placements leads to an efficient operation (albeit increased by a $1/\tau$ factor), even in the unbalanced case.

4.2 More Details and the Gaps

This section describes further algorithmic details, notably how gaps are laid out and the full insertion algorithm.

Chunk states

The presence of small buffers poses a challenge in the analysis, primarily due to roundoff errors (floors and ceilings) aggregating across multiple levels. We avoid this issue by excluding small buffers using an additional chunk state, described next. This detail allows us to amortize the rounding error against more elements.

We associate with each chunk a state that is either BUFFERED or UNBUFFERED. As the name suggests, UNBUFFERED chunks have no buffer; BUFFERED chunks may contain some buffer bounded by the buffer space invariant.[5] The states toggle as follows. An empty chunk is initially UNBUFFERED. When the chunk's size grows to $N(c_i) \geq 2/\tau^2$, it becomes BUFFERED. When the chunk's nonbuffer space drops below $N(c_i) < 1/\tau^2$, it becomes UNBUFFERED.

In summary, a chunk with $N(c_i) < 1/\tau^2$ is always UNBUFFERED, a chunk with $N(c_i) \geq 2/\tau^2$ is always BUFFERED, and a chunk with $1/\tau^2 \leq N(c_i) < 2/\tau^2$ may be either BUFFERED or UNBUFFERED depending on which threshold it crossed more recently.

Insertions and deletions

Insertions operate as follows; see Figure 4. This code includes handling of gaps, which we ignore for now in our discussion, i.e., ignore lines 10–16 and line 19, and assume $Z = Y$ in line 17. To insert an element into a district (level-0 chunk c_0), if the corresponding level-0 buffer has nonzero size $B(c_0) > 0$, replace the leftmost buffer slot with the element being inserted, thereby decreasing the buffer size.

If $B(c_0) = 0$, on the other hand, then a ***rebalance*** occurs, which consist of cascading chunk ***rebuild***s, starting at the level-0 chunk c_0. In general, an (insertion-triggered) rebuild of a chunk c_i causes c_i to increase its total space by taking empty space from its parent's buffer, and moving that empty space to the right end of c_i's buffer. If the parent's buffer is not large enough to handle the child's space request, the parent must first be rebuilt. In this way, the rebuild may cascade through the nearest ancestor chunks of c_0, whose buffers are all located to the right of c_0. The rebalance ensures that after the insertion, all rebuilt BUFFERED ancestor chunks c_i have the ***desired buffer size*** $\lfloor \tau N'(c_i)/2 \rfloor$, where N' denotes the sizes after the operation completes.

More precisely, a rebuild (on insertion) takes as argument a level-i chunk c_i and some number X of additional slots to be given to a nested child, with $X = 1$ when rebuilding a level-0 chunk (to accommodate the newly inserted element). Since the child is to add X slots to its total space, we have $N'(c_i) = N(c_i) + X$ after the operation; hence the desired buffer size is $d = \lfloor \tau(N(c_i) + X)/2 \rfloor$ as long as the chunk is BUFFERED. The rebuild of c_i takes $Y = d - B(c_i) + X$ slots from its parent, if available. If not, the parent is rebuilt first with requested space $Z = Y$. After taking space from the parent, $S(c_i)$ increases by Y slots, all of which are initially stored in the buffer. After this recursion, each chunk has collected all the space that it needs and rebuilds, leaving the final buffer size matching the desired buffer size.

We now consider how c_i takes Z space from its parent. Without gaps, all of the space in question belongs to the parent's buffer, which is located somewhere to the right of c_i. If c_i is the left child of its parent, then this space can be taken by sliding the entire intervening right level-i chunk to the right by Z slots. If c_i is a right child, then the parent's buffer is contiguous with it, and the empty buffer slots can simply be reassigned to c_i.

Figure 3 shows an example for insertion and rebalance, where all chunks are BUFFERED. An insert into the leftmost cursor district has insufficient space in the level-0 buffer (a). There is also insufficient space in the level-1 buffer to rebuild the level-0 buffer, so the rebalance propagates to the next level. First, the level-1 buffer is rebuilt by moving the sibling level-1 chunk (including 2 districts) to the right to move space from the level-2 buffer. Then the level-0 buffer is rebuilt by moving the sibling district to the right. In general, a rebalance may propagate through all H levels, requiring a buffer at each level to be rebuilt. At the end, the rebuilt buffers (in this case at levels 0 and 1) have buffer sizes equaling $\lfloor \tau N'(c)/2 \rfloor$.

Deletions are similar, but instead of taking slots from the parent's buffer, slots are returned to the parent's buffer.[6]

```
1: procedure INSERT(x, j)      // add x to jth district
2:     Let c_0 be level-0 chunk containing the jth district.
3:     if B(c_0) = 0 then REBUILD(c_0,1)
4:     Insert x in first empty buffer slot.

5: procedure REBUILD(c_i, X)
       // X ≥ 1 is the number of slots taken by a child
6:     if N(c_i) + X ≥ 2/τ^2 then mark c_i as BUFFERED
       // d is desired buffer size
7:     d = 0
8:     if c_i is BUFFERED then d = ⌊τ(N(c_i) + X)/2⌋
9:     Y = (d − B(c_i)) + X      // Y is increased to S(c_i)
10:    if c_i is a left chunk then
11:        g_p = G(parent(c_i))
12:        TAKE leftmost min{g_p, Y} level-(i+1) gaps
           from parent(c_i).
13:        Z = Y − g_p
           // Z is the number of level-(i+1) buffer slots to take
14:    else // c_i is a right chunk
15:        Calculate g, the number of level-(i+1) gaps
           distributed throughout the Y new space.
16:        Z = Y + g
17:    if Z > B(parent(c_i)) then REBUILD(parent(c_i), Z)
18:    TAKE leftmost Z slots from parent(c_i)'s buffer.
19:    Tag g of the slots as level-(i+1) gaps if c_i is a right chunk.
```

Figure 4: Pseudocode for insert and rebuild in the k-cursor data structure.

[5]Note that BUFFERED does not mean that $B(c_i) > 0$. A BUFFERED chunk may have $B(c_i) = 0$.

[6]In fact, deletions are more straightforward as the parent can be rebuilt *after* the slots are returned to it.

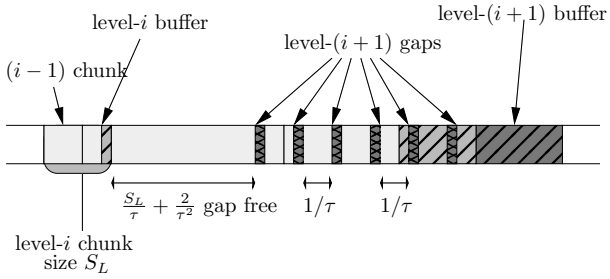

Figure 5: The recursive layout of a single level-$(i+1)$ chunk with gaps present, including a left level-i chunk followed by a right level-i chunk followed by a level-$(i+1)$ buffer. As with Figure 2, a second level of the recursive layout is displayed. Level-$(i+1)$ gaps are spread throughout the right level-i chunk, with $1/\tau$ slots between each gap. The leftmost S_L gaps (have been used and) are not present, where S_L is the total size of the left level-i chunk.

When deleting the last element in a district (level-0 chunk c_0), the slot it occupies is "returned" to the corresponding level-0 buffer. If the level-0 buffer size now violates the buffer space invariant, i.e., $B(c_0) > \tau N(c_0)$ where $N(c_0)$ is the number of postdeletion elements, then it is rebuilt to the same desired buffer size $d = \lfloor \tau N(c_0)/2 \rfloor$ by returning space to its parent. If the nonbuffer size is too small, i.e., $N(c_0) < 1/\tau^2$, then the chunk becomes UNBUFFERED and $d = 0$. In general, a level-i buffer is rebuilt to the desired buffer size by returning the excess elements to the nearest level-$(i+1)$ buffer to the right and sliding the entire intervening level-i sibling (if there is one) to the left. This process is analogous to the insertion rebuild. If the rebuild causes the level-$(i+1)$ parent chunk c_{i+1}'s buffer to exceed its threshold (i.e., either $B(c_{i+1}) > \tau N(c_{i+1})$ or $N(c_{i+1}) < 1/\tau^2$), then it should also be rebuilt in a similar manner. After the rebalance completes, all rebuilt buffers of BUFFERED chunks have buffer space exactly $\lfloor \tau N(c_i)/2 \rfloor$, as with insertions.

Gap placement

Consider a level-$(i+1)$ chunk c_{i+1}. The level-$(i+1)$ gaps, each a single slot in the array, are incorporated into only the right level-i chunk c_R as shown in Figure 5 according to the following invariant. There are no level-$(i+1)$ gaps in the left level-i chunk. Here is where we require $1/\tau$ to be an integer, and we choose τ later to satisfy this integrality assumption.

INVARIANT 11. *(Gap invariant) Consider a level-$(i+1)$ chunk c_{i+1} containing left and right level-i chunks c_L and c_R, respectively. The leftmost level-$(i+1)$ gap appears after the first $2/\tau^2 + S(c_L)/\tau$ slots of the right level-i chunk c_R. Another gap appears after each of the next $1/\tau$ slots in c_R.*

The gap invariant is important from both directions. Specifically, the insert analysis requires that gaps not be too far from the left chunk so that a scan does not need to travel far to find empty space. The delete analysis, on the other hand, requires that gaps not be too near the left chunk so that gaps can be inserted into the right chunk without scanning very far. The gap invariant also implies the space invariant on gaps (i.e., $G(c_{i+1}) \le \tau S(c_R)$). The additive $2/\tau^2$ term implies that UNBUFFERED chunks contain no gaps, which is convenient in the analysis.

Recall that each level-$(i+1)$ gap is counted towards the nonbuffer space $N(c_{i+1})$ and the total space $S(c_{i+1})$ for the chunk c_{i+1}, but these gaps are not counted towards a child's space even though they interleave with the right chunk. When recursively considering the layout or rebuilding of the contained level-i chunks, think of the level-$(i+1)$ or higher gaps as being elided—the position of level-$(i+1)$ gaps are only manipulated when considering the level-$(i+1)$ chunk. This elision is only for understanding the algorithm—the analysis must cope with the fact that "skipping" the gaps introduce additional cost.

How gaps affect updates in the k-cursor structure

We discuss how the insertion procedure copes with gaps; see Figure 4. The variable X should now be interpreted as the number of slots to add to the child's buffer plus the number of level-i gaps in c_i that this additional space requires. Since these gaps count towards c_i's nonbuffer space, it is still true that $N(c_i)$ is to increase by X, leaving the desired buffer space unaffected. Thus, Y denotes the total increase to $S(c_i)$ as before. We next consider two cases, depending on whether c_i is a left child or right child of its parent. Some handling of the cases is combined in the pseudocode, but we consider each case separately from start to finish here.

Suppose c_i is a left child, and let c_R be its right sibling. When rebuilding c_i, it should not contain any level-$(i+1)$ gaps. Hence, c_i need only take Y empty level-$(i+1)$ slots from its parent. The difference is how it takes that space. Rather than drawing only from its parent's buffer, c_i first consumes the leftmost g_p level-$(i+1)$ gaps that are spread throughout c_R. Taking these gaps means sliding some prefix of c_R to the right to fill in the appropriate previously empty spaces.[7] If there are not enough level-$(i+1)$ gaps to handle c_i's space request, i.e., $Y > g_p$, then c_i takes the remaining $Z = Y - g_p$ level-$(i+1)$ slots from its parent's buffer. If the parent does not have that many, it must first be rebuilt.

Suppose c_i is a right child, and let c_L be its left sibling. In this case, there are no level-$(i+1)$ gaps to the right of c_i's buffer, and all the space c_i takes from its parent comes from the parent's buffer. But adding Y space to c_i may require introducing up to $\lceil \tau Y \rceil$ level-$(i+1)$ gaps, according to the gap invariant. It is straightforward to calculate the number g of new gaps given the total space $S(c_i)$, the total space of its left neighbor $S(c_L)$, and the amount Y of increase to $S(c_i)$. These gaps would be counted towards the parent's nonbuffer space, so the space needed from the parent's level-$(i+1)$ buffer is increased to $Z = Y + g$. Once appropriated, all of these empty slots appear at the end of c_i's buffer, but g of them (spread evenly, according to the gap invariant) belong to c_i's parent as level-$(i+1)$ gaps and should be tagged appropriately (line 19).

Deletions may be similarly augmented to handle gaps. When rebuilding a left level-i chunk to return slots to its parent, return them as either level-$(i+1)$ gaps or buffers according to the gap invariant. Gaps can be introduced to the right sibling by sliding the right sibling to the left to consume the returned space. When returning space from a right level-i chunk to its parent, also return any level-$(i+1)$ gaps that are embedded in that space.

[7]Note that taking these gaps leaves the nonbuffer space of the parent unaffected, and hence this step does not risk violating the invariant on buffer space.

4.3 Analysis

This section analyzes the space usage and provides a sketch of the performance analysis.

Space analysis and prefix density

We now prove claims on the number of array slots used by regions of the k-cursor sparse array.[8] These claims are useful for proving both the desired prefix density and bounds on the cost of rebuilding buffers. We show that the data structure achieves constant prefix density. Throughout this section, we set $\tau = \frac{\delta'}{\lceil \lg k \rceil + 1}$, for some $0 < \delta'$ to be defined later in terms of τ so that $1/\tau$ is an integer greater than $\lceil \lg k \rceil + 1$.

The following bounds the space used by any chunk. Recall that slots counted towards a level-i chunk include those for all level-$(\leq i)$ buffers and gaps, but not higher-level gaps, so these slots are not necessarily contiguous.

LEMMA 12. *A level-i chunk c_i with a total of x elements in all descendent districts is assigned at most $S(c_i) \leq (1 + 3\tau)^{i+1}x$ array slots.*

PROOF. We proceed by induction on level i. For the base case, a level-0 chunk corresponds to a district with x elements and its size $\leq \tau x$ buffer, for at most $(1 + \tau)x$ slots. There are no level-0 gaps.

For the inductive step, consider a level-i chunk with $i > 0$. Let x_L and x_R be the number of elements in the left and right level-$(i - 1)$ chunks, respectively, and let S_L and S_R be the total number of slots used by these chunks. Then the space used by the level-i chunk is at most $(S_L + S_R)$ plus any level-i gaps plus the level-i buffer. By the gap-space invariant we have $G(c_i) \leq \tau S_R \leq \tau(S_L + S_R)$. The nonbuffer space is then upper bounded by $N(c_i) \leq (1 + \tau)(S_L + S_R)$, which implies the buffer space is at most $\tau(1 + \tau)(S_L + S_R)$ by the buffer-space invariant. Adding the nonbuffer space to the buffer space, we have $S(c_i) \leq (1 + \tau)^2(S_L + S_R) \leq (1 + 3\tau)(S_L + S_R)$ for $\tau \leq 1$. By inductive assumption, we have $S_L + S_R \leq (1 + 3\tau)^i x_L + (1 + 3\tau)^i x_R = (1 + 3\tau)^i x$. We thus conclude that the level-i chunk uses at most $(1 + 3\tau)(1 + 3\tau)^i x = (1 + 3\tau)^{i+1}x$ space. □

We now extend the above lemma for a particular choice of τ:

COROLLARY 13. *Let $\tau = \frac{\delta'}{H+1}$ and $H = \lceil \lg k \rceil$, where δ' is chosen from the range $0 < \delta' \leq 1/6$. Then a chunk with x elements in all descendent districts is stored in at most $(1 + 6\delta')x$ array slots.*

PROOF. The worst case space usage occurs at the highest level (level H). According to Lemma 12, the space is at most $(1 + 3\tau)^{H+1}x = (1 + \frac{3\delta'}{H+1})^{H+1}x$ space. We then observe that $(1 + \frac{3\delta'}{H+1})^{H+1} \leq e^{3\delta'} = \sum_{j=0}^{\infty} \frac{(3\delta')^j}{j!} < \sum_{j=0}^{\infty}(3\delta')^j$. For $0 < \delta' \leq 1/6$, we have $\sum_{j=0}^{\infty}(3\delta')^j = \frac{1}{1-3\delta'} \leq 1 + 6\delta'$. Multiplying by x gives the corollary. □

The preceding lemma and corollary ignore higher-level gaps within each chunk. The following lemma and corollary bound the number of gaps in a contiguous subarray.

LEMMA 14. *Any contiguous region of s slots in the array contains at most $\lceil \tau s \rceil$ level-i gaps, for any level i. Moreover, the first s slots in the array contain at most $\lfloor \tau s \rfloor$ level-i gaps.*

PROOF. By the gap invariant, there are $1/\tau$ slots allocated to a nested level-$(i - 1)$ chunk between each level-i gap. This fact is true even when the region spans different level-i chunks, as the first gap in the chunk does not appear until at after least $1/\tau$ slots. The addition of higher-level gaps can only cause the distance between two gaps in the array to increase. We conclude that there are at least $1/\tau$ array slots between any two level-i gaps, and hence the total number of level-i gaps in the region is at most $\lceil \tau s \rceil$.

When the region corresponds to a prefix of the array, the first gap begins after at least $2/\tau^2 \geq 1/\tau$ slots, which completes the proof. □

COROLLARY 15. *Let $\tau = \frac{\delta'}{H+1}$ and $H = \lceil \lg k \rceil$, where $\delta' > 0$. Any contiguous region of s slots in the array contains at most $\delta' s + (H + 1)$ gaps across all levels. Moreover, the first s slots in the array contain at most $\delta' s$ gaps.*

PROOF. Sum Lemma 14 across all $H + 1$ levels with $\lceil \tau s \rceil$ rounded up to $\tau s + 1$. □

The following theorem shows that our k-cursor sparse array guarantees constant prefix density. Setting $\delta' \leq \delta/9$ yields a space bound of $(1 + \delta)x$ array slots to store the first x elements. To satisfy the integrality requirement on τ, we choose $\delta' = \frac{1}{\lceil 9/\delta \rceil}$.

THEOREM 16. *Let $\tau = \frac{\delta'}{H+1}$ and $H = \lceil \lg k \rceil$, where δ' is chosen from the range $0 < \delta' \leq 1/6$. Then the earliest x elements in the k-cursor sparse array are stored within the first $(1 + 9\delta')x$ array slots.*

PROOF. First, observe that any prefix of the array consists of a sequence of complete chunks of decreasing level, followed by at most one partial district.[9] Ignoring any higher-level gaps, a complete chunk with y elements uses $(1 + 6\delta')y$ space from Corollary 13. Ignoring higher-level gaps, a partial district contains only real elements, and thus uses y space to store y elements. Summing across all these chunks, we have $(1 + 6\delta')x$ space.

To incorporate higher-level gaps, we apply Corollary 15. Specifically, the first s slots in the array contains at most $\delta' s$ gaps. It follows that the first s slots include the first x elements as long as $s - \delta' s \geq (1+6\delta')x$, or $s \geq \frac{(1+6\delta')}{1-\delta'}x$. With $\delta' \leq 1/6$, we have $\frac{(1+6\delta')}{1-\delta'} < 1 + 9\delta'$, and hence $(1 + 9\delta')x$ slots must contain at least x elements. □

Performance analysis

We now outline the analysis for the amortized cost of insertions and deletions. We focus on achieving an $O(\log^4 k/\delta^2)$ amortized cost per operation, with a brief discussion about removing a $\delta \log k$ factor. Proving the $O(\log^3 k/\delta^3)$ bound entails coping with some additional complications (notably rounding error). The full details, including the $O(\log^3 k/\delta^3)$ analysis, are deferred to the full version.

Note that the reason for chunk states (BUFFERED and UNBUFFERED) is also to facilitate the better analysis. Throughout this section, instead consider a simplified version of the data structure where every chunk is always BUFFERED.

Our analysis is an accounting argument—we associate money with particular substructures, and we argue that

[8] All of these claims are implicitly intended to apply after fully processing some sequence of operations—the bounds may temporarily be violated during a rebalance event.

[9] E.g., the first 11 districts correspond to a level-3 chunk followed by a level-1 chunk followed by a level-0 chunk.

enough money is released to pay for the cost of any rebuild. Throughout this section, we implicitly adopt the terminology/variables (i.e., X, Y, Z) used in Figure 4.

We first consider the actual cost of each rebuild:

LEMMA 17. *Consider a* BUFFERED *chunk c_i that is being rebuilt to increase or decrease its space usage by $Y \geq 1$ slots. Then the cost of the rebuild is $O(Y/\tau^2)$.*

PROOF SKETCH. There are several cases to consider (insert or delete on a left or right chunk). Since the parent's buffer is contiguous, the most expensive case is when c_i is a left child and the Y slots taken (or returned) are gaps. From the gap invariant, these Y slots are located within the next $2/\tau^2 + S(c_i)/\tau + Y/\tau$ slots of the parent. From Corollary 15 and the fact that $H < 1/\tau$, the cost of scanning that many slots in the parent is at most a constant factor more.

To complete the proof sketch, we claim that whenever a BUFFERED chunk c_i is rebuilt, we must have $Y = \Omega(\tau S(c_i))$. Combining this claim with the above bound, the dominating term is $O(S(c_i)/\tau) = O(Y/\tau^2)$. Again, there are several cases. For an insert, the idea is that a chunk is only rebuilt if its buffer would be more than drained, and hence the number of slots Y it takes is at least $\lceil \tau S(c_i)/2 \rceil$. □

Our accounting argument pins a specific amount of money to each level-i chunk. More precisely, we define currencies $\$_i$ at each level in the data structure. For any chunk, we maintain the invariant that c_i has at least

$$\$_i \left| \widehat{B}(c_i) - B(c_i) \right| ,$$

where $B(c_i)$ is the current buffer space and $\widehat{B}(c_i)$ is the buffer just after the previous rebalance the rebuilt c_i.

The value of a level-i dollar $\$_i$ depends on i. Specifically, a level-i dollar is worth the following number of normal \$'s:

$$\$_i 1 = \$(H + 1 - i) \left(1 + \frac{4}{H+1}\right)^{H+1-i} , \qquad (1)$$

where $H = \lceil \lg k \rceil$. We will charge $\Theta(1/\tau^2) = \Theta(\log^2 k/\delta^2)$ units of work to each dollar. Note that level-0 dollars are the most valuable, each worth $\$_0 1 \leq \$(H+1)e^4 = \$\Theta(\log k)$, or $\Theta(\log^3 k/\delta^2)$ units of work.

The main idea of the analysis is to leverage the difference in values across levels. In particular, consider when a level-i chunk c_i is rebuilt (on insert), requesting Y slots from its parent. During the rebuild, $\widehat{B}(c_i)$ changes and after the rebalance $\widehat{B}(c_i) = B(c_i)$. Thus, c_i's entire account can be used for the rebuild. The challenge is that slots are taken from c_i's parent's buffer, so the parent must be compensated. The key observation, following from Equation 1, is that currency can be converted across levels at the following rate:

$$\$_i 1 \quad \geq \quad \$1 + \$_{i+1} \left(1 + \frac{4}{H+1}\right) . \qquad (2)$$

In this way, if c_i has $\$_i D$, then we can afford $\$D$ for the rebuild itself, and also pass $\$_{i+1}(1 + \frac{4}{H+1})D$ to the parent.

The remaining argument is to show that c_i has enough money stored to pay for its rebuild and to compensate its parent. The argument proceeds by induction over rebuilds from low to high: We assume inductively that c_i has $\$_i D$, where D is at least the number of buffer slots already consumed plus the number X of slots requested by the rebuilding child; we prove that c_i can afford to pass $\$_{i+1} Z$ to its parent. To do so, we argue that $Z \leq (1+O(\tau))D + O(1) \leq (1 + \frac{4}{H+1})D + O(1)$, which entails applying relationships among different types of space. (The constant 4 in Equation 1 was specifically chosen to exceed the one in $O(\tau)$.) From the conversion rate, $\$_i D$ yields $\$_{i+1}(1+\frac{4}{H+1})D \geq \$_{i+1} Z - \$_{i+1} O(1)$, which almost fully pays the parent (except for $\$_{i+1} O(1)$). Moreover, $D = \Omega(Z)$ and hence $D = \Omega(Y)$, so by Lemma 17 the $\$D$ given off by the conversion are also enough to pay for the rebuild itself. We charge the $\$_{i+1} O(1)$ at each rebuilt level to the insertion itself, giving a total cost of at most $\$_0 O(H) = \$O(\log^2 k) = O(\log^2 k/\tau^2) = O(\log^4 k/\delta^2)$.

The preceding argument suffers from rounding errors that may occur at all H levels of granularity. A more sophisticated argument leveraging the chunk states allows us to amortize most of those $\$_{i+1} O(1)$'s against $\Theta(1/\tau^2)$ insertions, requiring each insert to pay at most $\$_0 O(1) = O(\log^3 k/\delta^2)$ work. (Our current analysis has a potentially unnecessary extra $1/\delta$ factor due to some extra boundary condition on chunk states.) To summarize the main result:

THEOREM 18. *The amortized cost of an insertion or deletion into the k-cursor sparse table is $O(\log^3 k/\delta^3)$.*

We next specialize the bound to the reallocation scheduler of Section 2 by considering the "charging pattern" and lost slots. Specifically, the next theorem directly implies Property 2. There are two main ideas to the proof, which is deferred to the full version. First, all money is passed only up to ancestors. Rebuilds of a chunk touch only space to the right of the chunk, so as money moves up it pays only for rebuilds that occur even further to the right. Second, to get an $O(1)$ bound on lost slots per district, let us consider what happens when rebuilding a left level-i chunk takes (or returns) Z slots from the parent. Essentially, Z empty space is moved to the left (or to the right), causing any district boundary in the right subtree to move rightward (or leftward) by up to Z slots; thus these districts lose at most $2Z$ slots. As we have $\$_i \Omega(Z)$ to work with, we need only charge $O(1)$ lost slots against each level-i dollar. Considering the path of money in the structure, a district can only charge to an insert when that insert's money is at a specific ancestor.

THEOREM 19. *A total of $O(\log^3 k/\delta^3)$ lost slots are charged to each insert or delete. Each lost slot is only charged to an operation occurring in a district to the left. There are at most $O(1)$ lost slots in each district charged to a particular insert or delete.*

Creating more cursors

If k is not known *a priori*, cursors can be added at the end of the structure without increasing the asymptotic costs. These additions must be at the end—the data structure does not support arbitrary cursor insertions. The one complication is that if k changes, then τ changes. Fortunately, we can modify the data structure to define τ locally—any chunk containing districts $\leq \ell$ uses $L = \lceil \lg \ell \rceil$ and hence $\tau' = \Theta(\delta/(\lceil \lg \ell \rceil + 1))$. None of the performance theorems are asymptotically affected by this change.

5. CONCLUSIONS

This paper has presented an efficient cost-oblivious reallocation algorithm for the sum of completion times. It remains open whether constant reallocation cost is possible; there are no nontrivial lower bounds. More generally, there

exist a wealth of unexplored reallocation problems in combinatorial optimization and scheduling, problems that arise when online decisions can be changed at some cost.

6. REFERENCES

[1] M. Andrews, M. X. Goemans, and L. Zhang. Improved bounds for on-line load balancing. *Algorithmica*, 23(4):278–301, 1999.

[2] C. Archetti, L. Bertazzi, and M. G. Speranza. Reoptimizing the Traveling Salesman Problem. *Networks*, 42(3):154–159, 2003.

[3] C. Archetti, L. Bertazzi, and M. G. Speranza. Reoptimizing the 0-1 knapsack problem. *Disc. Appl. Math.*, 158(17):1879–1887, Oct. 2010.

[4] G. Ausiello, V. Bonifaci, and B. Escoffier. Complexity and approximation in reoptimization. In S. B. Cooper and A. Sorbi, editors, *Computability in Context: Computation and Logic in the Real World*, pages 101–129. World Scientific, 2011.

[5] G. Ausiello, B. Escoffier, J. Monnot, and V. T. Paschos. Reoptimization of minimum and maximum traveling salesman's tours. *J. Disc. Alg.*, 7(4):453–463, 2009.

[6] G. Baram and T. Tamir. Reoptimization of the minimum total flow-time scheduling problem. In G. Even and D. Rawitz, editors, *Proc. MedAlg*, volume 7659 of *LLNCS*, pages 52–66, 2012.

[7] M. A. Bender, M. Farach-Colton, S. P. Fekete, J. T. Fineman, and S. Gilbert. Reallocation problems in scheduling. In *Proc. SPAA*, pages 271–279, 2013.

[8] M. A. Bender, M. Farach-Colton, S. P. Fekete, J. T. Fineman, and S. Gilbert. Cost-oblivious storage reallocation. In *Proc. PODS*, pages 278–288, 2014.

[9] M. A. Bender and H. Hu. An adaptive packed-memory array. *Trans. Datab. Syst.*, 32(4), 2007.

[10] H.-J. Böckenhauer, L. Forlizzi, J. Hromkovic, J. Kneis, J. Kupke, G. Proietti, and P. Widmayer. Reusing optimal TSP solutions for locally modified input instances. In *Proc. TCS*, pages 251–270, 2006.

[11] J. Bulánek, M. Koucký, and M. Saks. Tight lower bounds for the online labeling problem. In *Proc. STOC*, pages 1185–1198, 2012.

[12] A. Caprara, L. Galli, L. Kroon, G. Maróti, and P. Toth. Robust train routing and online re-scheduling. In *Proc. ATMOS*, pages 24–33, 2010.

[13] V. Chiraphadhanakul and C. Barnhart. Robust flight schedules through slack re-allocation. *EURO Journal on Transportation and Logistics*, 2(4):277–306, 2013.

[14] S. Davis, J. Edmonds, and R. Impagliazzo. Online algorithms to minimize resource reallocations and network communication. In *Proc. APPROX-RANDOM*, pages 104–115, 2006.

[15] L. Epstein and A. Levin. A robust APTAS for the classical bin packing problem. In *Proc. ICALP*, pages 214–225, 2006.

[16] L. Epstein and A. Levin. Robust algorithms for preemptive scheduling. In *Proc. ESA*, pages 567–578, 2011.

[17] S. P. Fekete, T. Kamphans, N. Schweer, C. Tessars, J. C. van der Veen, J. Angermeier, D. Koch, and J. Teich. Dynamic defragmentation of reconfigurable devices. *ACM Trans. Reconf. Technol. Syst.*, 5(2):8:1–8:20, June 2012.

[18] R. Graham, E. Lawler, J. Lenstra, and A. Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Ann. Disc. Math.*, 5:287 – 326, 1979.

[19] N. G. Hall and C. N. Potts. Rescheduling for new orders. *Op. Res.*, 52(3), 2004.

[20] A. Itai and I. Katriel. Canonical density control. *IPL*, 104(6):200–204, 2007.

[21] A. Itai, A. G. Konheim, and M. Rodeh. A sparse table implementation of priority queues. In *Proc. ICALP*, pages 417–431, 1981.

[22] H. Jiang and C. Barnhart. Dynamic airline scheduling. *Transp. Sc.*, 43(3):336–354, 2009.

[23] D. Karger, C. Stein, and J. Wein. *Scheduling Algorithms*. CRC Press, 1998.

[24] I. Katriel. Implicit data structures based on local reorganizations. Master's thesis, Technion, May 2002.

[25] P. Kouvelis and G. Yu. *Robust Discrete Optimization and Its Applications*. Kluwer, 1997.

[26] S. Lan, J.-P. Clarke, and C. Barnhart. Planning for robust airline operations: Optimizing aircraft routings and flight departure times to minimize passenger disruptions. *Transp. Sc.*, 40(1):15–28, 2006.

[27] J. M. Mulvey, R. J. Vanderbei, and S. A. Zenios. Robust optimization of large-scale systems. *Op. Res.*, 43(2), 1995.

[28] P. Sanders, N. Sivadasan, and M. Skutella. Online scheduling with bounded migration. *Math. Oper. Res.*, 34(2):481–498, 2009.

[29] H. Shachnai, G. Tamir, and T. Tamir. A theory and algorithms for combinatorial reoptimization. In *Proc. LATIN*, pages 618–630, 2012.

[30] M. Skutella and J. Verschae. A robust PTAS for machine covering and packing. In *Proc. ESA*, pages 36–47, 2010.

[31] C. A. Tovey. Rescheduling to minimize makespan on a changing number of identical processors. *Nav. Res. Logist.*, 33:717–724, 1986.

[32] A. T. Unal, R. Uzsoy, and A. S. Kiran. Rescheduling on a single machine with part-type dependent setup times and deadlines. *Ann. Op. Res.*, 70, 1997.

[33] J. C. Verschae. *The Power of Recourse in Online Optimization Robust Solutions for Scheduling, Matroid and MST Problems*. PhD thesis, TU Berlin, June 2012.

[34] J. Westbrook. Load balancing for response time. *J. of Alg.*, 35(1):1 – 16, 2000.

[35] D. Willard. Maintaining dense sequential files in a dynamic environment (extended abstract). In *Proc. STOC*, pages 114–121, 1982.

[36] D. E. Willard. Good worst-case algorithms for inserting and deleting records in dense sequential files. In *Proc. SIGMOD*, pages 251–260, 1986.

[37] D. E. Willard. A density control algorithm for doing insertions and deletions in a sequentially ordered file in good worst-case time. *I&C*, 97(2):150–204, 1992.