The Dissertation Committee for Yu Sun
certifies that this is the approved version of the following dissertation:

Reconfigurable Resource Scheduling

Committee:

Greg Plaxton, Supervisor

Sanjoy Baruah

Aloysius Mok

Lili Qiu

Harrick Vin

Reconfigurable Resource Scheduling

by

Yu Sun, B.E., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2007

To my parents

Acknowledgments

First of all, I am truly indebted to my advisor Greg Plaxton. His enthusiasm towards problem solving and insistence on elegance have constantly inspired me throughout the past few years. His guidance and contribution have been invaluable in making this dissertation possible and better. All the results in this dissertation are coauthored with Greg.

Some other members of my dissertation committee also contribute to this dissertation. Harrick Vin is a co-author of the work presented in this dissertation. His enlightening insights, valuable comments, and kind support help improve the dissertation, and make it more accessible. Sanjoy Baruah has shown great interest and enthusiasm in the work presented in the dissertation, and has provided encouraging and insightful questions that help us improve the presentation in chapter 4. I am also thankful to Aloysius Mok and Lili Qiu for their time and valuable comments made at my oral proposal.

I am grateful to the fellow graduate student Mitul Tiwari for many illuminating discussions. He is also a co-author of the results presented in the dissertation. I am also thankful to our graduate coordinator Gloria Ramirez, who is always efficient and kind to help.

I am deeply indebted to my parents, who never fail to support and believe in me. This dissertation is dedicated to them.

Reconfigurable Resource Scheduling

Publication No. ____________________

Yu Sun, Ph.D.
The University of Texas at Austin, 2007

Supervisor: Greg Plaxton

Multi-core and multi-processor environments are increasingly used to support a wide range of applications. These environments host multiple services simultaneously. The set of processors configured to support a particular service depends upon the associated workload; fluctuations in workload require changes in processor allocation. In these systems, reallocating a processor from one service to another tends to incur a nonnegligible overhead. Motivated by these applications, this dissertation considers a class of scheduling problems that we refer to as *reconfiguration resource scheduling*. The salient features of this class are as follows: There are jobs of different categories, and resources can be reconfigured to process jobs of a certain category, where a reconfiguration incurs an overhead, in terms of cost or time.

In our initial investigation, we study the following subclass of the class of reconfigurable resource scheduling problems. We are given a finite set of resources, each of which has an associated category, and a sequence of requests,

each of which is a set of unit jobs. Each job has an associated category, and needs to be executed on a resource of the same category within a specified delay bound of its arrival, or else it is dropped at a specified drop cost. At any time, a resource can be reconfigured to a different category at a specified reconfiguration cost. The goal is to schedule the reconfigurations of the resources, and the executions of the jobs, in a way that minimizes the total cost.

We design efficient online algorithms with provably good performance for two main problems in this subclass, one allowing category-specific drop costs, which we refer to as *reconfigurable resource scheduling with variable drop costs*, and the other allowing category-specific delay bounds, which we refer to as *reconfigurable resource scheduling with variable delay bounds*.

Reconfigurable resource scheduling with variable drop costs is motivated by certain applications in which some jobs are more important than others. We solve this problem using a layered approach, where in each layer we reduce to a scheduling problem defined over a more constrained set of possible inputs. In the first layer, we reduce to the special case in which all job arrivals are batched. In the second layer, we reduce to the special case in which the job arrival rate is limited. In the third layer, we reduce the rate-limited problem to two cases: large reconfiguration cost, and small reconfiguration cost. We use a traffic reshaping technique to smooth out the job arrivals, and thereby reduce the case with large reconfiguration cost to the special case of unit delay, and reduce the case with small reconfiguration cost to the special case of rate-limited unit delay. In the fourth layer, we reduce unit delay with large reconfiguration cost to a caching problem which we refer to as file caching with remote reads, and reduce rate-limited unit delay with small reconfiguration cost to a variant of disk paging problem which we refer to as prefix paging. In the fifth layer, we solve the file caching with remote reads problem by generalizing certain existing work in the area of file caching, and we solve prefix paging using a kind of marking algorithm.

Reconfigurable resource scheduling with variable delay bounds is motivated by applications in which jobs are required to be processed within category-specific delay guarantees. Once again, we use a layered approach. The first two layers are analogous to the first two layers in our solution for reconfigurable resource scheduling with variable drop costs, respectively, but are more involved due to the variable delay bounds. In the third layer, we solve the rate-limited problem using a novel combination of the EDF and LRU scheduling principles.

Table of Contents

Chapter 1

Introduction

This dissertation addresses a class of scheduling problems referred to as reconfigurable resource scheduling. Problems in this class arise in certain emerging network applications that involve dynamically allocating a large number of shared resources to a variety of services. The primary goal of this dissertation is to design online algorithms for problems in this class with provably good performance across a wide range of operating conditions. Such algorithms are valuable in practice due to the wide range of operating conditions that exist in various applications, the rapidly evolving nature of network applications, and the inherent difficulty in modifying a scheduling algorithm designed for one application to meet the needs of another.

1.1 Background and Motivation

Multi-core and multi-processor environments are increasingly used to support a wide range of high-throughput applications, such as web services, network applications, and database servers. These environments host multiple services simultaneously (e.g., a router supporting various packet processing services).

To isolate — with respect to security and performance — services from one another, these environments often configure processors to support only one service at a time. The set of processors configured to support a particular service depends upon the associated workload; fluctuations in workload require changes in processor allocation. For instance, a shared data center dynamically adjusts the allocation of processors to independent services as the composition of the workload changes [9, 10]. Similarly, a multi-service router based on multi-core network processors adjusts the allocation of processors to different packet categories as the traffic load fluctuates [29, 30, 32]. In these systems, reallocating a processor from one category to another tends to incur a nonnegligible overhead. For instance, on Intel's IXP2400 network processor, loading the instruction store of a processor core with the code for a new category incurs a context switch time, which is much (two or three orders of magnitude) greater than the time to process a packet [16]. In certain applications involving QoS guarantees, jobs are required to be processed within a delay tolerance, where the delay tolerance is a function of the job category [17].

Motivated by the aforementioned applications, this dissertation considers a class of scheduling problems that we refer to as *reconfiguration resource scheduling*. The salient features of this class are as follows: (1) there are jobs of different categories; (2) resources can be reconfigured to process jobs of a certain category, where a reconfiguration incurs an overhead, in terms of cost or time.

1.2 Contributions and Techniques

Most problems related to scheduling and resource allocation require the algorithm to operate in an *online* manner, that is, to make irrevocable decisions in response to each incoming request, with no knowledge of the future request sequence. Our high level goal in this line of research is to provide robust, self-tuning online algorithms that provide provably good performance across a wide range of operating conditions. Such algorithms are valuable in practice since reconfigurable resource scheduling problems can arise in different scenarios and applications, and it is inherently difficult to modify a scheduling algorithm designed for one application to meet the needs of another. We adopt the framework of competitive analysis (see 2.1 for a detailed discussion of competitive analysis), in which the performance of an online algorithm is measured against that of an optimal offline algorithm, that is, an algorithm that knows all the future requests.

As an initial exploration, we study a subclass of the class of reconfigurable resource scheduling problems within the framework of competitive analysis. The following is an informal description of the subclass; a formal definition is given in Chapter 2.2. We are given a finite set of resources, each of which has an associated category, and a sequence of requests, each of which is a set of unit jobs. Each job has an associated category, and needs to be executed on a resource of the same category within a specified delay bound of its arrival, or else it is dropped at a specified drop cost. At any time, a resource can be reconfigured to a different category at a specified reconfiguration cost. The goal is to schedule the reconfigurations of the resources, and the executions of the jobs, in a way that minimizes the total cost.

In this dissertation, we solve two main problems in the aforementioned subclass: (1) one with a fixed delay bound, a fixed reconfiguration cost, and category-specific drop costs, which we refer to as *reconfigurable resource scheduling with variable drop costs*; (2) the other with category-specific delay bounds, a fixed reconfiguration cost, and a fixed drop cost, which we refer to as *reconfigurable resource scheduling with variable delay bounds*. We establish formal results for the two problems in Chapters 3 and 4, respectively. (The preliminary versions of these results appear in [23] and [24], respectively.) In solving the two main problems, we consider some special cases and intermediate problems, some of which may be of independent interest. For example, file caching with remote reads, discussed in Section 3.3, is a generalization of the file caching problem studied by Irani [12] and Young [33], and a special case of the k-server problem with excursions [19].

In the following, we highlight the main techniques that we use to solve the above two reconfigurable resource scheduling problems.

1.2.1 A Layered Approach

In our initial investigation of reconfigurable resource scheduling problems, we would like to determine which problems in this class admit online algorithms that are optimal up to constant factors. We find it convenient to adopt a layered approach, where each successive layer reduces to a scheduling problem defined over a more constrained set of possible inputs. The layered approach enables us to attack a complicated problem by solving several simpler ones, although the layered approach has a tendency to build up constant factors. The key ideas underlying some of the layers are useful in solving various problems in the reconfigurable resource scheduling class. For example, in both of our solutions to reconfigurable resource scheduling with variable drop costs and reconfigurable resource scheduling with variable delay bounds, we have a layer which reduces the main problem to a special case in which job arrivals are batched, which simplifies the problem by reducing the unpredictability of the request sequence.

1.2.2 Traffic Reshaping

One source of the difficulty in solving reconfigurable resource scheduling problems is the potential burstiness in the traffic (i.e., job arrivals). It is not uncommon that network applications use traffic regulating schemes, e.g., leaky bucket [31], to control the volume of the incoming traffic. In solving reconfigurable resource scheduling with variable drop costs, we use a *reshaping* scheme as a way to smooth out the job arrivals and eliminate the delay bound parameter.

Our reshaping scheme maps each job to a unit time interval between its arrival time and deadline, and requires each job to either to be executed at the time to which it is mapped, or to be dropped. The mapping is computed in the following local manner: We partition the sequence of requests into "frames", where a frame is the sequence of requests corresponding to an integral multiple of the specified delay bound, and map the jobs that appear in each frame independently of those appearing in other frames.

1.2.3 Exploiting Connections to Paging Problems

A well-studied problem similar to reconfigurable resource scheduling is the disk paging problem [28]. Disk paging considers a two-level memory system: the slow memory that can store a set of fixed-size pages P, and the fast memory that can store a subset of pages in P. Given a request for a page $p \in P$, if page p is not in the fast memory (called a miss), the system must load p into the fast memory. Given a request sequence, the goal is to minimize the total number of misses. The pages of the fast memory in a paging problem are analogous to the resources in a reconfigurable resource scheduling problem. Loading a page of the fast memory is analogous to reconfiguring a resource with a particular category.

The fundamental difference is that in disk paging, one request for a page arrives at a time, whereas in reconfigurable resource scheduling, multiple jobs, of one or more categories, can arrive at a time. Therefore, in the context of reconfigurable resource scheduling, it may be necessary to configure several resources to process jobs with the same category. There are two additional differences between a disk paging problem and a reconfigurable resource scheduling problem considered in this dissertation. First, in a disk paging problem, a request is required to be served immediately, whereas in a reconfigurable resource scheduling problem considered in this dissertation, a request does not need to be served immediately, but is requiring to be executed within a specified delay bound to avoid any penalty. Second, in a disk paging problem, a requested page is required to be loaded into the fast memory, whereas in a reconfigurable resource scheduling problem considered in this dissertation, a request can be dropped by paying a penalty.

The file caching problem studied by Irani [12] and Young [33] is a generalization of the disk paging problem in which different files (the counterpart of pages) may have different sizes and retrieval costs. We are able to reduce reconfigurable resource scheduling with variable drop costs to a generalization of the file caching problem, in which on a miss, we have an option to read the file remotely instead of requiring it to be loaded into the cache. We refer to this problem as *file caching with remote reads*.

Young proposes the *Landlord* algorithm to solve the file caching problem. The main idea of the *Landlord* algorithm is to maintain a real-valued credit for each file in the cache, and to use the credit to indicate when a file should be evicted from the cache. We solve file caching with remote reads by modifying *Landlord* and its associated analysis. The main modification is that we maintain a credit for each file (not only those in the cache), and we use the credit to decide when to load a file into the cache and when to evict a file from the cache.

1.2.4 Exploiting Connections to Scheduling Paradigms

Many scheduling problems are solved by traditional scheduling principles such as EDF (Earliest Deadline First), LSF (Least Slack First), and LRU (Least Recently Used). To attack reconfigurable resource scheduling problems, it is natural to attempt to make use of these traditional scheduling principles. In the context of reconfigurable resource scheduling with variable delay bounds, it seems that algorithms based on one of the the above scheduling principles suffers from either *thrashing* (excessive reconfiguration cost) or *underutilization* (excessive drop cost), and therefore fails to provide a good solution.

Though EDF alone or LRU alone seems insufficient to solve reconfigurable resource scheduling with variable delay bounds, each maintains a dynamic ordering that addresses a key aspect of the request sequence: EDF addresses the urgency aspect, and LRU addresses the recency aspect. Motivated by this observation, we propose a novel and efficient combination of EDF and LRU. The main idea is to keep two sets of categories configured, one selected by the EDF principle, and one selected by the LRU principle. We prove that this combination yields an online algorithm within a constant factor of optimal. This result suggests that, for problems which cannot solved by a single traditional scheduling principle, it is worthwhile to explore the combination of two or more traditional scheduling principles.

1.3 Related Work

In this section, we discuss other work in scheduling and power management that is relevant to the class of reconfigurable resource scheduling problems studied in this dissertation. Additional work related to specific issues in the context of reconfigurable resource scheduling with variable drop costs and variable delay bounds is presented in the relevant technical chapters.

Scheduling. Brucker [5, Chapter 9] surveys a class of offline scheduling problems in which each job belongs to a certain group, and between the executions of any two jobs in different groups on the same machine, there is a changeover time, during which the machine cannot process any job. Results for single and multiple machine problems with changeover time are summarized. For a variant with identical machines, equal sized groups, and equal processing and changeover times, Brucker et al. [6] give a polynomial time offline algorithm that decides whether there exists a schedule in which all jobs are executed within a common delay bound.

In a recent position paper, Srinivasan et al. [30] discuss the scheduling problems that arise in multi-core network processors, and consider the application of existing multiprocessor scheduling algorithms in this domain. Various challenges are pointed out, and some initial ideas towards addressing these concerns are presented. Kokku [16] proposes a scheduling algorithm, called Everest, for multi-core network processors. The parameters considered are per-service delay bounds, per-service execution requirements, and a fixed context switch time. The primary goal is to maximize the number of packets processed within a service-specific delay tolerance. Everest is shown to perform well in experiments.

Another related scheduling problem is "scheduling with rejection" [3, 26, 27]. In this problem, jobs can be rejected at a certain cost. The objective is to minimize the sum of (1) the makespan of the schedule for the executed jobs, and (2) the total cost of the rejected jobs. Constant competitive algorithms are given for both nonpreemptive and preemptive versions of the problem.

Power Management. Two main schemes have been used to minimize power usage in battery-operated embedded systems: sleep state and dynamic speed scaling [13]. Power management with sleep state exploits the ability to put a resource into sleep state when idle. In the sleep state, the resource consumes less power, but a certain energy is required to transition the resource to the active state, in which jobs can be processed on the resource. Such problems can be viewed as problems in the framework of reconfigurable resource scheduling in which it is possible to reconfigure a resource into a sleep state. For power management problems in which each resource has multiple sleep states, Irani et al. [14] give deterministic algorithms that consume at most twice the amount of energy as the optimal offline algorithm. In their work, it is assumed that the transitions between different states occur instantaneously. Ramanathan et al. [25] perform an experimental study that discusses the tradeoff between optimizing for latency and power in this context.

1.4 Outline of Dissertation

The remainder of this dissertation is organized as follows. In Chapter 2, we give a detailed discussion of the competitive analysis, and make formal definitions of the reconfigurable resource scheduling problems considered in this dissertation. In Chapter 3, we present our solution to reconfigurable resource scheduling with variable drop costs. In Chapter 4, we present our solution to reconfigurable resource scheduling with variable delay bounds. Finally we make some concluding remarks in Chapter 5.

Chapter 2

Preliminaries

2.1 Competitive Analysis

In competitive analysis, we seek algorithms that achieve a good *competitive ratio* [28], that is, the maximum ratio between the cost incurred by the online algorithm and that incurred by an optimal offline algorithm, over all request sequences. (Informally, an online algorithm achieves a good competitive ratio if for any request sequence, its performance is close to that of an optimal offline algorithm. For a comprehensive introduction to online computation and competitive analysis, see the textbook by Borodin and El-Yaniv [4].)

A drawback of competitive analysis is that its worst case mindset can be overly pessimistic. For example, in their seminal paper on competitive analysis [28], Sleator and Tarjan prove that the competitive ratio of any online paging algorithm is k, where k is the number of pages in the cache. This is an extremely negative result. However, Sleator and Tarjan observe that if the offline algorithm is given only h pages, for some $h \leq k$, then the competitive ratio can be improved to $\frac{k}{k-h+1}$, and that this optimal competitive ratio is achieved by LRU as well as a number of other simple online paging algorithms. For example, this result says that if the online algorithm is given a factor of two advantage in the size of its cache, then it will achieve performance within a factor of two of the optimal offline algorithm. This relaxed version of competitive analysis, in which the online algorithm is given extra resources, is later referred to as *resource augmentation* [15, 22], and can be viewed as a method to compensate the online algorithm for its lack of future information. We refer to an online algorithm that achieves a constant competitive ratio when given a constant factor resource advantage as a *resource competitive* algorithm.

2.2 Problem Definitions

Before we define the reconfigurable resource scheduling problems considered in this dissertation, we first make some preliminary definitions. We define a *request* as a (possibly empty) set of unit *jobs*, where each job is characterized by a non-black *color*, a nonnegative integer *arrival time*, a positive integer *delay bound*, and a positive integer *drop cost*. The *deadline* of a job is defined as the arrival time plus the delay bound. There is a finite set of *resources* on which jobs are executed. For convenience, the resources are numbered from 0. At any time, each resource has an associated color. There is a reconfiguration cost to reconfigure a resource, i.e., to change the color of a resource.

The processing of a given request sequence σ proceeds in rounds numbered from 0 to $|\sigma| - 1$. At the beginning of round i, we have a set of *pending* jobs, each of which has an arrival time smaller than i, and a deadline at least

$i + 1$. Each round i consists of four phases: (1) in the first phase, the *arrival phase*, request i is received; (2) in the second phase, the *reconfiguration phase*, each resource can be reconfigured to a different color; (3) in the third phase, the *execution phase*, each resource configured with color ℓ can execute at most one pending job of color ℓ; (4) in the fourth phase, the *drop phase*, pending jobs with deadline $i + 1$ are dropped.

For convenience, we view each resource has a sequence of *slot*s, where slot i corresponds to round i. We order the slots in increasing order of resource indices, breaking ties by slot indices. We say a slot is *free* if no job is executed in the slot, and *occupied* otherwise.

Before defining a schedule, we find it technically convenient to define a pseudo-schedule. Throughout most of this dissertation, we use the notion of a schedule instead of a pseudo-schedule. In Section 3.7, we find it useful to make use of the notion of a pseudo-schedule. Given a request sequence σ, a *pseudo-schedule* decides, for each job x in σ, whether to execute x or not, and if so, on which resource and in which round. A *coloring* maps each resource to a color. Given an initial coloring μ, the minimum-cost set of reconfigurations made by a pseudo-schedule P can be deduced from μ and P. Therefore, at times, we find it convenient to allow the reconfigurations to be specified implicitly; at other times, for the purpose of analysis, we find it is more convenient to explicitly specify the reconfigurations. We define the number of resources used by a pseudo-schedule as the number of resources that are reconfigured at least once. A pseudo-schedule is allowed to use an arbitrary number of resources.

A *schedule* is a pseudo-schedule that respects a certain bound on the number of resources.

Consider any request sequence σ and any pseudo-schedule P for σ. We use $DropCost(P)$ to denote the drop cost incurred by P. Given an initial coloring μ, we use $ReconfigCost(P, \mu)$ to denote the reconfiguration cost incurred by P. Given an initial coloring μ, we define $Cost(P, \mu)$ as the sum of $DropCost(P)$ and $ReconfigCost(P, \mu)$. In this dissertation, if the initial coloring is not specified, we assume the *default coloring* in which all resources are *black*. Note that the color of any job is not black. For the reconfigurable resource scheduling problems addressed in this dissertation, the objective is to devise a schedule S for σ such that $Cost(S)$ is minimized.

For the reconfigurable resource scheduling problems considered in this dissertation, the input is a pair (σ, m), where σ is a request sequence, and m is a positive integer. Given an instance (σ, m), an algorithm produces a schedule for σ. An algorithm is said to be *offline* if it knows all the requests in advance, and it is said to be *online* if it makes irrecoverable decisions without knowing the future requests. An algorithm A is *b-feasible* if for any instance (σ, m), A produces a schedule that uses at most $b \cdot m$ resources. An algorithm is *feasible* if it is 1-feasible. For any instance (σ, m) and any algorithm A, the cost (resp., reconfiguration cost, drop cost) of A on (σ, m), denoted $Cost(A, \sigma, m)$ (resp., $ReconfigCost(A, \sigma, m)$, $DropCost(A, \sigma, m)$), is defined as $Cost(S)$ (resp., $ReconfigCost(S)$, $DropCost(S)$), where S is the schedule produced by A on (σ, m). An algorithm A is *(a, b)-competitive* if A is b-feasible and for any instance (σ, m), $Cost(A, \sigma, m)$ is at most $a \cdot Cost(OPT, \sigma, m)$, where OPT is an optimal feasible offline algorithm. An algorithm A is *resource competitive* if A is (a, b)-competitive for some positive reals a and b.

The focus of this dissertation is to give resource competitive online algorithms for some problems in the class of reconfigurable resource scheduling. To refer to these problems in a convenient manner, we introduce the [*reconfig* | *drop* | *delay* | *batch*] notation. The *reconfig* field describes the details of the reconfiguration cost. In this dissertation, the possible values for this field are: a fixed reconfiguration cost, denoted Δ; and per-color reconfiguration costs, denoted Δ_ℓ. (Throughout this dissertation, we follow the convention that the symbol ℓ is used to denote a color. Hence, the symbol Δ_ℓ indicates that the reconfiguration cost depends on color ℓ.) The *drop* field describes the details of the drop cost. In this dissertation, the possible values for this field are: a unit drop cost, denoted 1; and per-color drop costs, denoted d_ℓ. The *delay* field describes the details of the delay bound. In this dissertation, the possible values for this field are: a unit delay bound, denoted 1, a fixed delay bound, denoted D; and per-color delay bounds, denoted D_ℓ. The *batch* field constrains the arrival rounds of requests of color ℓ to occur at integral multiples of the specified value. In this dissertation, the possible values for this field are 1, D, and D_ℓ. With this notation, the problem of reconfigurable resource scheduling with variable drop costs is denoted $[\Delta \mid d_\ell \mid D \mid 1]$. The problem of reconfigurable resource scheduling with variable delay bounds is denoted $[\Delta \mid 1 \mid D_\ell \mid 1]$. In this dissertation, we also use this notation to specify some intermediate problems that we consider in solving the above two main problems.

Chapter 3

Reconfigurable Resource Scheduling with Variable Drop Costs

3.1 Introduction

This chapter presents our solution to reconfigurable resource scheduling with variable drop costs, that is, $[\Delta \mid d_\ell \mid D \mid 1]$. We give a resource competitive algorithm for this problem, where the competitive ratio that we obtain does not depend on the various problem parameters, that is, D, Δ, and the d_ℓ's.

We solve this problem with a layered approach. First, we use *batching* to reduce the main problem to the special case in which jobs arrive at integral multiples of D, denoted $[\Delta \mid d_\ell \mid D \mid D]$. Second, we reduce the latter problem to two cases: (1) $\Delta < d_\ell$, for each color ℓ, and (2) $\Delta \geq d_\ell$, for each color ℓ. We use a *reshaping* technique to reduce the two cases to two intermediate reconfigurable resource scheduling problems in which $D = 1$, denoted $[\Delta_\ell \mid d_\ell \mid 1 \mid 1]$, where $\Delta_\ell \geq d_\ell$ for each color ℓ, and rate-limited $[\Delta \mid d_\ell \mid 1 \mid 1]$, where $\Delta < d_\ell$ for each color ℓ, respectively. As the notation suggests, for the case where $D = 1$, we actually solve a more general variation that allows per-color reconfiguration costs Δ_ℓ, as long as $\Delta_\ell \geq d_\ell$ for each color ℓ. Third, we use a *serializing* technique to reduce $[\Delta_\ell \mid d_\ell \mid 1 \mid 1]$, where $\Delta_\ell \geq d_\ell$ for each color ℓ, to a file caching problem that we refer to as file caching with remote reads, and reduce rate-limited $[\Delta \mid d_\ell \mid 1 \mid 1]$, where $\Delta < d_\ell$ for each color ℓ, to a variant of the disk paging problem which we refer to as prefix paging. File caching with remote reads generalizes the file caching problem studied by Irani [12] and Young [33], and we solve it by modifying Young's *Landlord* algorithm. We use a kind of marking algorithm to solve the problem of prefix paging.

The intuition underlying each layer is as follows. The first layer reduces the unpredictability of the request sequence. The second layer smooths out job arrivals and eliminates the delay bound parameter. The third layer reduces the job arrival rate.

Throughout this chapter, we make use of the following definitions. For any nonnegative integer i, we define *block* (resp., *half-block*) i as the D (resp., $\frac{D}{2}$) rounds starting with round $i \cdot D$ (resp., $i \cdot \frac{D}{2}$).

The remainder of this chapter is organized as follows. Section 3.2 discusses other work related to the specific issues addressed in this chapter. Section 3.3 presents our solution to file caching with remote reads. Section 3.4 presents our solution to $[\Delta_\ell \mid d_\ell \mid 1 \mid 1]$, where $\Delta_\ell \geq d_\ell$ for each color ℓ. Section 3.5 presents our solution to prefix paging. Section 3.6 presents our solution to rate-limited $[\Delta \mid d_\ell \mid 1 \mid 1]$, where $\Delta < d_\ell$ for each color ℓ. Sections 3.7 and 3.8 present our solution to rate-limited $[\Delta \mid d_\ell \mid D \mid D]$. Section 3.9 presents our solution to $[\Delta \mid d_\ell \mid D \mid D]$. Section 3.10 presents our solution to the main problem $[\Delta \mid d_\ell \mid D \mid 1]$.

3.2 Related Work

Paging and File Caching Problems. In Section 3.3, we define and solve a file caching problem which we refer to as file caching with remote reads, as a building block of our solution to reconfigurable resource scheduling with variable drop costs. File caching with remote reads is a generalization of the file caching work by Irani [12] and Young [33], which themselves can be viewed as generalizations of the work in the classic disk paging problem studied by Sleator and Tarjan [28]. Irani [12] proposes an counter-based randomized online algorithm that is shown to be $O(\log^2 k)$, where k is the ratio of the size of the cache to the size of the smallest file; Young [33] proposes an algorithm, called *Landlord*, that is shown to be $\frac{n-m+1}{m}$, where n and m are sizes of online and offline caches, respectively. The main idea of *Landlord* is to maintain a credit for each file in the cache; on a miss, if the cache is full, "rent" is charged to each file in the cache proportional to its size, and the files that run out of credit are evicted. Cao and Irani [8] propose an algorithm called GreedyDual-Size, which is similar to *Landlord*, and show that GreedyDual-Size performs well in experiments. Our algorithm for file caching with remote reads is obtained by modifying *Landlord*.

Some other work related to our file caching problem includes the k-server problem with excursions and page migration problem. Manasse et al. [19] consider the k-server problem with excursions, in which a request can be satisfied remotely by a server or it can be satisfied by moving any server to the requested vertex. In page migration related problems, the requested page can either be accessed remotely, or, it can be moved to the requesting processor. Some closely related work in this realm is k-page migration considered by Bartal et al. [2], and constrained page migration considered by Albers and Koga [1]. In k-page migration, the system maintains k copies of any page, and the local memories have unlimited capacity. In the constrained page migration problem, local memories have limited capacity. There are some similarities between the DLRU algorithm of Albers and Koga and our file caching algorithm presented in Section 3.3. However, the DLRU algorithm does not provide a solution to our file caching problem.

Traffic Shaping. A traffic regulator like leaky bucket [31] reduces the burstiness in the network traffic. In our solution to reconfigurable resource scheduling with variable drop costs, we use a reshaping technique to map each job to a specific round, which can be viewed as a way to reduce the burstiness in the request sequence.

3.3 File Caching with Remote Reads

In this section, we introduce a new caching problem, referred to as file caching with remote reads, as a building block within the overall solution to our main problem. This problem is similar to the file caching problem studied by Irani [12] and Young [33]. The difference is that, on a miss, a remote read can be issued to serve the request instead of writing the requested file to the cache. We modify the *Landlord* algorithm and its associated analysis given by Young to solve our caching problem.

3.3.1 Problem Definition

We are given a universal set of files and a cache. Each file x is characterized by a positive integer size, denoted by $size(x)$; a nonnegative read cost, denoted by $read(x)$; a nonnegative write cost, denoted by $write(x)$. The input is a pair (σ, m), where σ is sequence of requests, each of which is a file (the file to be accessed), and m is an integer that indicates the bound on the cache size. Initially, the cache is empty. To process a request x, an algorithm can first perform an arbitrary long sequence of the following two actions: removing files from the cache with no cost, and writing the requested file x into the cache with cost $write(x)$, provided there is sufficient room. Then, if x is in the cache, the algorithm incurs no further cost. Otherwise, the algorithm performs a remote read, paying $read(x)$. The goal is to maintain the files in the cache so as to minimize the total cost.

3.3.2 Algorithm *LLL*

We present a *Landlord*-like algorithm, denoted *LLL*, as follows. For each file x, maintain a real value $credit(x)$ (whether x is in the cache or not). Initially the credit of any file is zero. On a request x, augment $credit(x)$ in the following way:

$$credit(x) := \min(credit(x) + read(x), write(x)).$$

When *credit*(x) reaches *write*(x), if x is not in the cache, repeatedly run the eviction procedure (to be described) until there is room for x in the cache, and then add x to the cache.

The eviction procedure is as follows. Charge every file in the cache rent until at least one file runs out of credit. More formally, for each file x in the cache, decrease *credit*(x) by $\delta \cdot \mathit{size}(x)$, where δ denotes the minimum credit per unit size of any file in the cache. Evict from the cache any nonempty subset of the files with zero credit.

3.3.3 Analysis of *LLL*

Before presenting the analysis, let us first introduce some definitions. Consider an arbitrary instance (σ, m) of the file caching with remote reads problem. We say that an algorithm for file caching with remote reads is feasible if the algorithm respects the bound on the cache size. Let *OFF* denote an arbitrary feasible offline algorithm. Let A and L denote the caches of *OFF* and *LLL*, respectively. By the definition of a feasible algorithm, the size of A is m. Let n ($n > 2m$) denote the size of L. Let $\mathit{Cost}(\mathit{OFF}, \sigma, m)$ and $\mathit{Cost}(\mathit{LLL}, \sigma, m)$ denote the cost incurred by *OFF* and *LLL* on (σ, m), respectively. Let $\mathit{ReadCost}(\mathit{OFF}, \sigma, m)$ (resp., $\mathit{ReadCost}(\mathit{LLL}, \sigma, m)$) denote the read cost incurred by *OFF* (resp., *LLL*) on (σ, m). Let $\mathit{WriteCost}(\mathit{OFF}, \sigma, m)$ (resp., $\mathit{WriteCost}(\mathit{LLL}, \sigma, m)$) denote the write cost incurred by *OFF* (resp., *LLL*) on (σ, m).

We define a potential function

$$\Phi = m \sum_{x} credit(x) + (n - m + 1) \sum_{x \in A} (write(x) - credit(x)).$$

Initially, because the credit of any file is zero, and both caches are empty, the potential is zero. Because *LLL* maintains the invariant that $0 \leq credit(x) \leq write(x)$, the potential is always nonnegative.

To analyze the performance of *LLL*, we execute *LLL* alongside *OFF*. As in [33], we process each successive request with *OFF*, and then with *LLL*. We then observe the effect of each action on the potential.

Actions taken by *OFF* to serve a request x can be broken down into a sequence of steps, with each step being one of the following: *OFF* evicts a file from the cache; *OFF* writes x to the cache; *OFF* performs a remote read for x. Actions taken by *LLL* to serve a request to file x can be broken down into a sequence of steps, with each step being one of following: *LLL* augments the credit of x; *LLL* charges rent; *LLL* evicts a file from the cache to make room for x; *LLL* writes x to the cache; *LLL* performs a remote read for x. Note that the credit augmentation is always performed and performed first in serving any request.

For an arbitrary request x, the effect of each action taken to serve x on the potential is given in Lemma 3.3.1 through Lemma 3.3.6.

Lemma 3.3.1. *If OFF performs a remote read, or LLL writes a file into the cache, or LLL performs a remote read, Φ does not change.*

Proof. Since the contents of A as well as $credit(x)$, for any file x, do not change, Φ remains unchanged. $\square$

Lemma 3.3.2. *If OFF writes a file x to the cache, Φ increases by at most $(n-m+1) \cdot write(x)$.*

Proof. The first summation does not change. The second summation increases by at most $write(x)$ because $0 \leq credit(x) \leq write(x)$. Hence, Φ increases by at most $(n-m+1) \cdot write(x)$. $\square$

Lemma 3.3.3. *If LLL augments the credit of a file x that is not in A, Φ increases by at most $m \cdot read(x)$.*

Proof. The first summation increases by at most $read(x)$. Since $x \notin A$, the second term does not change. Hence, Φ increases by at most $m \cdot read(x)$. $\square$

Lemma 3.3.4. *If OFF evicts a file from the cache, Φ does not increase.*

Proof. The first summation does not change. The second summation does not increase since $write(x) \geq credit(x)$. Hence, Φ does not increase. $\square$

Lemma 3.3.5. *If LLL augments the credit of x that is in A, Φ decreases by at least $(n-2m+1) \cdot s \geq 0$, where $s \leq read(x)$. Also, if $s < read(x)$, LLL does not perform a remote read in serving x.*

Proof. By the way the credit is augmented on an access, the first summation increases by s, where $s \leq read(x)$. Also, if $s < read(x)$, after the credit augmentation, $credit(x)$ reaches $write(x)$, and *LLL* subsequently writes x into the cache and does not perform a remote read in serving x. Since $x \in A$, the second summation decreases by $(n-m+1) \cdot s$. Hence, Φ decreases by at least $(n-2m+1) \cdot s \geq 0$. $\square$

Lemma 3.3.6. *If LLL charges rent to make room for a file x, Φ does not increase.*

Proof. The potential Φ decreases by δ times $m \cdot \mathit{size}(L) - (n-m+1) \cdot \mathit{size}(L \cap A)$, where $\mathit{size}(X)$ denotes $\sum_{x \in X} \mathit{size}(x)$. Note that $\mathit{size}(L) > n - \mathit{size}(x) + 1$ and $\mathit{size}(L \cap A) \leq m$. Since $\mathit{size}(x) \leq m$, Φ decreases by at least $m \cdot (n-m+1) - (n-m+1) \cdot m = 0$. Hence, Φ does not increase. $\square$

Lemma 3.3.7. *For any instance (σ, m) of file caching with remote reads, the total increase of Φ is at most*

$$m \cdot \mathit{ReadCost}(\mathit{OFF}, \sigma, m) + (n-m+1) \cdot \mathit{WriteCost}(\mathit{OFF}, \sigma, m).$$

Proof. Consider the steps taken by *OFF* and *LLL* to serve a request x. By Lemmas 3.3.1 through 3.3.6, Φ increases only in the following two cases. In the first case, *OFF* writes x to the cache. By Lemma 3.3.2, Φ increases by at most $(n-m+1) \cdot \mathit{write}(x)$. In this case, the write cost incurred by *OFF* in serving x is at least $\mathit{write}(x)$. In the second case, *LLL* updates the credit of x that is not in A. By Lemma 3.3.3, Φ increases by at most $m \cdot \mathit{read}(x)$. In this case, the read cost incurred by *OFF* in serving x is $\mathit{read}(x)$. In either case, the increase of Φ in serving x is at most $m \cdot \mathit{ReadCost}(\mathit{OFF}, x) + (n-m+1) \cdot \mathit{WriteCost}(\mathit{OFF}, x)$. Summing up over all x, the lemma follows. $\square$

Lemma 3.3.8. *For any instance (σ, m) of file caching with remote reads, the total negative change of Φ is at least*

$$(n - 2m + 1) \cdot (\mathit{ReadCost}(\mathit{LLL}, \sigma, m) - \mathit{ReadCost}(\mathit{OFF}, \sigma, m)).$$

Proof. We focus our attention on an arbitrary request x for which LLL performs a remote read in serving x. Consider the steps taken by OFF and LLL to serve x. As indicated earlier, credit augmentation is always performed by LLL in serving any file. When LLL augments the credit of x, if x is in A, then Φ decreases by $(n - 2m + 1) \cdot \mathit{read}(x)$ by Lemma 3.3.5; otherwise, OFF incurs a read cost of $\mathit{read}(x)$ in serving x. In either case, $(n - 2m + 1) \cdot \mathit{ReadCost}(\mathit{LLL}, x, m)$ is at most the decrease of Φ in serving x plus $(n - 2m + 1) \cdot \mathit{ReadCost}(\mathit{OFF}, x, m)$, so the decrease of Φ in serving x is at least $(n - 2m + 1) \cdot (\mathit{ReadCost}(\mathit{LLL}, x, m) - \mathit{ReadCost}(\mathit{OFF}, x, m))$. Summing up over all such files x's, the lemma follows. $\square$

Lemma 3.3.9. *For any instance (σ, m) of file caching with remote reads,*

$$\mathit{WriteCost}(\mathit{LLL}, \sigma, m) \leq \mathit{ReadCost}(\mathit{LLL}, \sigma, m).$$

Proof. For any file x, we define an epoch as follows. An epoch of x ends the moment x is kicked out of the cache. A new epoch of x starts when the previous epoch ends. Fix any file x and any epoch i of x. By algorithm LLL, the credit of x at the beginning of epoch i is zero. In epoch i, before the credit reaches $\mathit{write}(x)$, for each access on x, the credit increases by at most $\mathit{read}(x)$, and algorithm LLL incurs a read cost of $\mathit{read}(x)$. When the credit reaches *write*(x), algorithm *LLL* writes x into the cache, incurring a write cost of *write*(x). After that, the algorithm does not incur any cost until epoch i ends. Hence, the write cost incurred by *LLL* during epoch i on x is at most the relevant read cost. Summing up over all files x, and all epochs i of x, the lemma follows. □

Theorem 3.1. *Algorithm LLL is* $\frac{2(n-m+1)}{n-2m+1}$*-competitive for file caching with remote reads.*

Proof. Consider an arbitrary instance (σ, m) of file caching with remote reads. By Lemmas 3.3.7 and 3.3.8, and the fact that Φ is always nonnegative, we have

$$\begin{aligned} & (n-2m+1)\cdot(\mathit{ReadCost}(\mathit{LLL},\sigma,m)-\mathit{ReadCost}(\mathit{OFF},\sigma,m)) \\ \leq \quad & m\cdot\mathit{ReadCost}(\mathit{OFF},\sigma,m)+(n-m+1)\cdot\mathit{WriteCost}(\mathit{OFF},\sigma,m). \end{aligned}$$

Since $n > 2m$, we have

$$\mathit{ReadCost}(\mathit{LLL},\sigma,m) \leq \frac{n-m+1}{n-2m+1}\cdot\mathit{Cost}(\mathit{OFF},\sigma,m).$$

The theorem follows from the above inequality and Lemma 3.3.9. □

3.4 Unit Delay

In this section we solve $[\Delta_\ell \mid d_\ell \mid 1 \mid 1]$, where $\Delta_\ell \geq d_\ell$ for all colors ℓ. Recall that this problem is characterized by per-color configuration costs Δ_ℓ, per-color drop costs d_ℓ, and a unit delay bound. As indicated earlier, our solution to this problem uses a reduction to file caching with remote reads, which is defined and solved in Section 3.3.

3.4.1 Algorithm *Serialize*

We first give some useful definitions. For an arbitrary request α for $[\Delta_\ell \mid d_\ell \mid 1 \mid 1]$, we define *serialized*$(\alpha)$ as a request sequence β obtained as follows. Each request in β is a file, and each file, denoted (ℓ, j), is characterized by a color ℓ, a nonnegative integer index j, a read cost d_ℓ, and a write cost Δ_ℓ. Let r_ℓ denote the number of color ℓ jobs in α. Let $X = \cup_\ell \{(\ell, j) \mid 0 \leq j < r_\ell\}$. We obtain β by ordering the files in X arbitrarily. It is not hard to see that *serialized*(α) is a request sequence for file caching with remote reads. For any request sequence σ for $[\Delta_\ell \mid d_\ell \mid 1 \mid 1]$, we define *serializedReqSeq*(σ) as a request sequence obtained from concatenating *serialized*(σ_i)'s, in increasing order of i, where σ_i is request i of σ.

Given an instance (σ, m) of $[\Delta_\ell \mid d_\ell \mid 1 \mid 1]$, where $\Delta_\ell \geq d_\ell$ for all colors ℓ, algorithm *Serialize* produces a schedule for σ in the following three stages. In the first stage, we use algorithm *LLL* (defined in Section 3.3.2) to obtain an n-resource schedule S' for $\sigma' =$ *serializedReqSeq*(σ), where $n = O(m)$.

In the second stage, we construct an n-resource schedule S'' for σ' as follows. For any nonnegative integer i, let S'_i be the portion of S' for $\sigma'_i =$ *serialized*(σ_i). We obtain a schedule S''_i for σ'_i by delaying the writes in S'_i to the beginning of σ'_{i+1}. We define schedule S'' as the concatenation of the S''_i's in increasing order of i. It is not hard to see that S'' is a schedule for σ'.

In the third stage, we construct an n-resource schedule S for σ as follows. Consider any nonnegative integer i. Consider any resource k. Let (ℓ, j) be the color of the file cached in location k right after S'' makes all the writes at the beginning of σ'_i in S''. In the reconfiguration phase of round i, we configure resource k with color ℓ. In the execution phase of round i, we execute as many jobs as the current configuration allows.

3.4.2 Analysis of *Serialize*

Lemma 3.4.1. *Consider an arbitrary instance (σ, m) of $[\Delta_\ell \mid d_\ell \mid 1 \mid 1]$, where $\Delta_\ell \geq d_\ell$ for all colors ℓ. If there exists an m-resource schedule T for σ with cost C, then there exists a schedule T' for serializedReqSeq(σ) with cost at most $4C$ and cache size m.*

Proof. For convenience of analysis, let σ_i be request i of σ, $\sigma'_i = \textit{serialized}(\sigma_i)$, and $\sigma' = \textit{serializedReqSeq}(\sigma)$. For any nonnegative integer i and any color ℓ, let $Y_{i,\ell}$ be the set of resources configured with color ℓ at the end of the reconfiguration phase of round i in T. The proof proceeds in three phases. First, for each nonnegative integer i, we construct an m-resource schedule T'_i for σ'_i, in increasing order of i. Second, we construct an m-resource schedule T' for σ' by concatenating T'_i's, in increasing order of i. Third, we bound the cost of T'. The second phase is straightforward. In the remainder of this proof, we offer the details of the first and third phases.

In the first phase, we construct schedule T'_i for σ'_i in the following two stages. In the first stage, for each color ℓ, we label the resources in $Y_{i,\ell}$ in round i as follows. If $i = 0$, we label the resources in $Y_{i,\ell}$ from 0 to $|Y_{i,\ell}| - 1$ arbitrarily. If $i > 0$, we proceed in the following phases. For any resource k in $Y_{i,\ell} \cap Y_{i-1,\ell}$ such that the label of resource k in round $i - 1$ is in the range $[0, |Y_{i,\ell}|)$, we let resource k inherit its label from round $i - 1$. We assign the remaining labels in $[0, |Y_{i,\ell}|)$ to the remaining resources in $Y_{i,\ell}$. In the second stage, we construct T'_i as follows. At the beginning of σ'_i, we configure the cache in the following manner. For any nonnegative integer k, if the color of resource k at the end of the reconfiguration phase of round i in T, call it ℓ, is black, then location k is empty; otherwise, location k caches page (ℓ, j), where j is the label assigned to resource k in round i. We maintain the above cache configuration until the end of σ'_i.

In the third phase, we bound $Cost(T')$ by showing the following two claims: (1) the write cost incurred by T' is at most four times the reconfiguration cost incurred by T; and (2) the read cost incurred by T' equals the drop cost incurred by T.

The proof of (1) proceeds in two stages. In the first stage, with each reconfiguration from color ℓ to color ℓ' in round i in T, we associate $\Delta_\ell + \Delta_{\ell'}$ units of credit. It is not hard to see that the total credit associated with the reconfigurations in T is twice the reconfiguration cost of T.

In the second stage, we need to show that the write cost incurred by T' is at most twice the total credit. Since T' only reconfigures the cache at the beginning of σ'_i's, T' only incurs write cost at the beginning of σ'_i's. Hence, it is sufficient to show that, for any nonnegative integer i, the write cost incurred by T' at the beginning of σ'_i is at most twice the credit associated with the reconfigurations made by T in round i, which we prove in the following three paragraphs.

Consider any nonnegative integers i and k. Consider any write operation W in T' at the beginning of σ'_i at location k, because of a reconfiguration operation R made by T on resource k in round i. It is easy to see that the write cost incurred by W is at most the credit associated with R. It is not hard to verify that, all write operations in T' at the beginning of σ'_0 corresponds to a reconfiguration operation made by T in round 0.

Consider any nonnegative integer $i > 0$. Consider the write operations made by T' because of the labeling of resources. Fix an arbitrary color ℓ. Let $p_{i,\ell}$ be the number of resources that change color from color ℓ or to color ℓ in round i in T. Let $q_{i,\ell}$ be the number of resources k such that resource k is configured with color ℓ at the beginning and throughout round i in T, and the label of resource k in round i is different from that in round $i-1$. It is easy to verify that the write cost incurred by T' at the beginning of σ'_i due to the relabeling of resources is $\sum_\ell q_{i,\ell} \cdot \Delta_\ell$. It is also easy to verify that the total credit associated with the reconfigurations from or to color ℓ made by T in round i is at least $\sum_\ell p_{i,\ell} \cdot \Delta_\ell$.

Let $Z_{i,\ell}$ be the set of resources that are configured with color ℓ and that have a label at least $|Y_{i+1,\ell}|$ in round i. By the way we assign labels to resources in each round, $q_{i,\ell}$ equals $|Z_{i-1,\ell}|$. It is straightforward to see that $|Z_{i-1,\ell}|$ is at most $\max(0, |Z_{i-1,\ell}| - |Z_{i,\ell}|)$, which in turn is at most $p_{i,\ell}$. Hence,

$q_{i,\ell} \le p_{i,\ell}$. Therefore, the write cost incurred by T' at the beginning of σ'_i is at most twice the total credit associated with reconfigurations made by T in round i. Summing up over all nonnegative integers i's, claim (1) follows.

The proof of (2) proceeds as follows. Consider any nonnegative integer i and any color ℓ. Let $r_{i,\ell}$ be the number of color ℓ jobs in σ_i. Let $p_{i,\ell}$ be the number of resources configured with color ℓ in T at the end of the reconfiguration phase of round i. So in round i, T pays a drop cost of $d_\ell \cdot \max(r_{i,\ell} - p_{i,\ell}, 0)$ on the color ℓ jobs in σ_i. By the definition of σ'_i, the set of color (ℓ, j) files, over all j, in σ'_i is $\{(\ell, j) \mid 0 \le j < r_{i,\ell}\}$. From the way we construct T', the set of color (ℓ, j) files, over all j, cached by T' at the beginning of σ'_i, and kept in the cache until the end of σ'_i, is $\{(\ell, j) \mid 0 \le j < p_{i,\ell}\}$. Hence, T' pays a read cost of $d_\ell \cdot \max(r_{i,\ell} - p_{i,\ell}, 0)$ on color (ℓ, j) files, over all j, in σ'_i. Hence the read cost incurred by T' on color ℓ files in σ'_i is at most the drop cost incurred by T' on the color ℓ jobs in σ_i. Summing up over all colors ℓ and all nonnegative integers i, claim (2) follows. □

Lemma 3.4.2. *Consider any instance (σ, m) of $[\Delta_\ell \mid d_\ell \mid 1 \mid 1]$, where $\Delta_\ell \ge d_\ell$ for all colors ℓ. Let $\sigma' = serializedReqSeq(\sigma)$. Let S' be the schedule produced by algorithm LLL on (σ', m), and S be the schedule produced by algorithm Serialize on (σ, m). Then $Cost(S) \le 2Cost(S')$.*

Proof. Let σ_i be request i of σ. Let S'' be the schedule produced in the second stage of algorithm *Serialize* on (σ, m). We bound $Cost(S)$ in the following two stages.

In the first stage, we establish that $Cost(S'') \leq 2Cost(S')$ as follows. We first bound the read cost of S''. Let S'_i and S''_i be the portion of S' and S'' for σ_i, respectively. Let X be the set of files that appear in σ'_i. By the definition of σ'_i, each file in X is unique. Let Y be the subset of X that consists of the files written into the cache in S'_i, and $Z = X \setminus Y$. For each file (ℓ, j) in Y, since each file is unique in X, S''_i incurs at most one read on (ℓ, j). Since $\Delta_\ell \geq d_\ell$, for all colors ℓ, the read cost incurred by S''_i on the set of files in Y is at most the write cost incurred by S'_i on the set of files in Y. It is not hard to see that the read cost incurred by S''_i on the set of files in Z equals the read cost incurred by S'_i on the set of files in Z. Hence, the read cost of S''_i is at most the read cost of S'_i. Summing up over all i, the read cost of S'' is at most the read cost of S'. Since S'' makes all the write operations made by S', the write cost of S'' equals that of S'.

In the second stage, we establish $Cost(S) \leq Cost(S'')$. Consider any nonnegative integer i. We proceed in the following two steps. In the first step, we show that the reconfiguration cost of S_i is at most the write cost of S''_i as follows. Consider any color ℓ. By the definition of S_i, and the way we construct S, a reconfiguration operation to color ℓ made by S_i corresponds to a write operation of a file (ℓ, j), for some j, made by S''_i. Since loading a cache location with a file (ℓ, k) by evicting a file (ℓ, j), $j \neq k$, in S'_i, incurs a write cost Δ_ℓ, the cost incurred by reconfigurations to color ℓ in S_i is at most the cost incurred by the writes of files (ℓ, j), over all j, in S''_i. Summing up over all colors ℓ, the reconfiguration cost of S_i is at most the write cost of S''_i.

In the second step, we show that the drop cost of S_i is at most the read cost of S''_i. Consider any color ℓ. Let $r_{i,\ell}$ be the number of color ℓ jobs in σ_i. By the definition of σ'_i, the set of color (ℓ, j) files, over all j, appearing in σ'_i is $\{(\ell, j) \mid 0 \leq j < r_{i,\ell}\}$. Let $p_{i,\ell}$ be the number of color ℓ files cached by S''_i right before the first request in σ'_i (i.e., the reconfigurations of the cache contents in S'_i at the beginning of σ'_i, if any, have been made). By the definition of S''_i and the way that we construct S'', S''_i does not change the cache configuration except at the beginning of σ'_i. So the read cost of S''_i on color (ℓ, j) files, over all j, is at least $d_\ell \cdot \max(r_{i,\ell} - p_{i,\ell}, 0)$. By the definition of S_i and the way we construct S, the number of color ℓ resources at the end of the reconfiguration phase of round i is also $p_{i,\ell}$, and S_i executes as many jobs as the current reconfiguration allows. Hence, the drop cost of S_i on color (ℓ, j) jobs, over all j, is $d_\ell \cdot \max(r_{i,\ell} - p_{i,\ell}, 0)$, which we have shown to be at most the read cost of S''_i on color ℓ jobs. Summing up over all colors ℓ, the drop cost of S_i is at most the read cost of S''_i. Summing up over all i, the claim in the second stage follows from the above two steps. □

Theorem 3.2. *Algorithm Serialize is resource competitive for* $[\Delta_\ell \mid d_\ell \mid 1 \mid 1]$, *where* $\Delta_\ell \geq d_\ell$ *for all colors* ℓ.

Proof. Consider any instance (σ, m) of $[\Delta_\ell \mid d_\ell \mid 1 \mid 1]$, where $\Delta_\ell \geq d_\ell$ for all colors ℓ. Suppose there exists an m-resource offline schedule T for σ with cost C. Let $\sigma' = \textit{serializedReqSeq}(\sigma)$. By Lemma 3.4.1, there exists a schedule T' for σ' with at most cost $4C$ and cache size m. By Theorem 3.1, algorithm *LLL* is resource competitive for file caching with remote reads. Hence the schedule S' produced by *LLL* on (σ', m) incurs a cost of $O(C)$ with a cache size of $O(m)$. By Lemma 3.4.2, the schedule S produced by algorithm *Serialize* on (σ, m) incurs a cost of $O(C)$. By definition, S uses the same number of resources as S'. Hence, the lemma follows. □

3.5 Prefix Paging

In this section, we define and solve a variant of the traditional disk paging problem; we refer to this variant as prefix paging. The input to the prefix paging problem is a pair (σ, m), where σ is a sequence of page requests, and m is an integer that denotes the bound on the cache for any feasible algorithm. Every page is identified by a pair (ℓ, j), where ℓ is a color and j is a nonnegative integer index in the range 0 to $m - 1$. The sequence σ is partitioned into contiguous segments of at most m requests each. The requests of a segment involve distinct pages and are presented in lexicographically sorted order. Within any given segment, the following prefix property holds: If there is a request (ℓ, j) where $j > 0$, then there is also a request $(\ell, j - 1)$. The rules for processing page requests are the same as in traditional disk paging.

3.5.1 Algorithm *Mark*

Given an arbitrary instance (σ, m) of the prefix paging problem, we partition the request sequence σ into epochs as follows. If fewer than $2m$ distinct pages are accessed in σ, then there is just one epoch. Otherwise, the first epoch is the shortest prefix p of σ such that the following two conditions hold: (1) p corresponds to a whole number of segments; (2) p contains accesses to at least $2m$ distinct pages. Having defined the first epoch, we define the rest of the epochs by recursively partitioning the remaining suffix of the request.

Our online algorithm, denoted *Mark*, which uses a cache of size $3m$, is a kind of marking algorithm, similar in spirit to the class of marking algorithms discussed, e.g., in [4, Section 3.5.1]. A mark bit is associated with each cache location. Initially, all cache locations are unmarked. During an epoch, a cache location that is read is marked, and remains marked until the beginning of the next epoch, at which point it is unmarked. If the cache is full and we suffer a miss, then an arbitrary page in an unmarked location is evicted. Note that such an unmarked location is guaranteed to exist, since the definitions of epoch and segment imply that, at all times, fewer than $3m$ cache locations are marked.

3.5.2 Analysis of *Mark*

Lemma 3.5.1. *After a page is accessed, it stays in the cache throughout the remainder of the processing of the current epoch.*

Proof. When a request for a page x is processed during a given epoch, the cache location from which x is read becomes marked, and remains marked until the end of the current epoch. Therefore, page x is not evicted before the end of the current epoch. $\square$

The following corollary is used in Section 3.6.

Corollary 3.5.1. *Immediately after processing a given segment, the cache contains all of the pages accessed during the segment.*

Proof. This is immediate from Lemma 3.5.1, since each epoch consists of a whole number of segments. □

Lemma 3.5.2. *Algorithm Mark is resource competitive for the prefix paging problem.*

Proof. By Lemma 3.5.1, during any epoch, algorithm *Mark* suffers at most one miss per distinct page accessed. By the definitions of epoch and segment, fewer than $3m$ distinct pages are accessed during an epoch. Thus algorithm *Mark* suffers fewer than $3m$ misses during any epoch.

Call an epoch *complete* if it contains accesses to at least $2m$ distinct pages, and *incomplete* otherwise. Note that at most one epoch is incomplete. Since the feasible offline algorithm has a cache size of m, there are at least $2m - m = m$ misses in each complete epoch.

Combining the results of the preceding paragraphs, we conclude that algorithm *Mark* is resource competitive on any instance with at least one complete epoch. It remains to consider instances consisting of a single incomplete epoch. Fix such an instance, and let k denote the number of distinct pages accessed. As argued earlier, algorithm *Mark* suffers at most k misses. Furthermore, any offline algorithm suffers at least k misses. So once again algorithm *Mark* is resource competitive. $\square$

3.6 Rate-Limited Unit Delay

In this section, we consider a special case of $[\Delta \mid d_\ell \mid 1 \mid 1]$, referred to as rate-limited $[\Delta \mid d_\ell \mid 1 \mid 1]$, where $\Delta < d_\ell$ for all colors ℓ, and for any instance (σ, m) of the special case, at most m jobs arrive per round in σ. Our algorithm for rate-limited $[\Delta \mid d_\ell \mid 1 \mid 1]$, where $\Delta < d_\ell$ for all colors ℓ, is invoked by algorithm *Split* in Section 3.7.2.

3.6.1 Algorithm *RLSerialize*

For an arbitrary request α for rate-limited $[\Delta \mid d_\ell \mid 1 \mid 1]$, where $\Delta < d_\ell$ for all colors ℓ, we define *serialized*(α) as in Section 3.4.1, except that we order the files in X lexicographically. For any request sequence σ for rate-limited $[\Delta \mid d_\ell \mid 1 \mid 1]$, where $\Delta < d_\ell$ for all colors ℓ, we define *serializedReqSeq*(σ) as in Section 3.4.1. It is not hard to see that *serializedReqSeq*(σ) is a request sequence for prefix paging.

Given an instance (σ, m) of rate-limited $[\Delta \mid d_\ell \mid 1 \mid 1]$, where $\Delta < d_\ell$ for all colors ℓ, algorithm *RLSerialize* produces a schedule for σ in the following two stages. In the first stage, we use algorithm *Mark* (defined in Section 3.5) to obtain a $3m$-resource schedule S' for $\sigma' =$ *serializedReqSeq*(σ).

In the second stage, we construct a $3m$-resource schedule S for σ from S' as follows. Consider an arbitrary nonnegative integer i. Let σ_i be request

i of σ. Consider any resource k, where $0 \le k < 3m$. Let (ℓ, j) be the page cached at location k immediately after serving the last request in $serialized(\sigma_i)$ in S'. In the reconfiguration phase of round i, we configure resource k with color ℓ in S. In the execution phase of round i, we execute as many jobs in σ_i as the current configuration allows.

3.6.2 Analysis of *RLSerialize*

We say that a schedule S is *drop-free* if S does not incur any drop cost.

Lemma 3.6.1. *Consider any instance (σ, m) of rate-limited $[\Delta \mid d_\ell \mid 1 \mid 1]$, where $\Delta < d_\ell$ for all colors ℓ. If there exists an m-resource schedule T for σ, then there exists an m-resource drop-free schedule T' for σ such that $Cost(T') \le 2Cost(T)$.*

Proof. We construct T' from T round by round. Consider any round i. We execute all the jobs that arrive in round i as follows: We execute the set of jobs executed in T on the same resources as in T; we also execute the set of jobs that are dropped in T on the resources that are idle in T, that is, resources on which no jobs are executed; note that there are always a sufficient number of idle resources, since at most m jobs arrive and need to be executed in a round.

It is straightforward to see that T' is drop-free. By executing a color ℓ job x dropped in T, T' reduces the drop cost by d_ℓ, and increases the reconfiguration cost by at most 2Δ. Since $\Delta < d_\ell$ for all colors ℓ, $Cost(T') \le 2Cost(T)$. $\square$

We omit the proof of Lemma 3.6.2 since it is analogous to the proof of Lemma 3.4.1, and is in fact simpler.

Lemma 3.6.2. *Consider any instance* (σ, m) *of rate-limited* $[\Delta \mid d_\ell \mid 1 \mid 1]$*, where* $\Delta < d_\ell$ *for all colors* ℓ*. If there exists an* m*-resource drop-free schedule* T *for* σ *with cost* C*, then there exists a schedule* T' *for serializedReqSeq*(σ) *that makes at most* $\frac{2C}{\Delta}$ *misses with cache size* m*.*

Lemma 3.6.3. *Consider any instance* (σ, m) *of rate-limited* $[\Delta \mid d_\ell \mid 1 \mid 1]$*, where* $\Delta < d_\ell$ *for all colors* ℓ*. Let* $\sigma' =$ *serializedReqSeq*(σ)*. Let* S' *be the* $3m$*-resource schedule produced by Mark on* (σ', m)*, and* S *be the* $3m$*-resource schedule produced by RLSerialize on* (σ, m)*. Then Cost*(S) *is at most* Δ *times the number of misses incurred by* S'*.*

Proof. Since for any color ℓ, we replace color (ℓ, j) in S', for any nonnegative integer j, with color ℓ in S, the reconfiguration cost incurred by S is at most Δ times the number of misses incurred by S'. It remains to show that S is drop-free.

Consider an arbitrary round i and color ℓ. Let σ_i be request i of σ and $\sigma'_i =$ *serialized*(σ_i). Let $X_{i,\ell}$ denote the set of color (ℓ, j) pages, over all j, in σ'_i. Let $Y_{i,\ell}$ denote the set of color (ℓ, j) pages, over all j, in the cache immediately after processing σ'_i in S'. Let $Z_{i,\ell}$ denote the set of resources configured with color ℓ at the end of the reconfiguration phase of round i in S. By the construction of S, $|Z_{i,\ell}| = |Y_{i,\ell}|$. By Corollary 3.5.1, pages in σ'_i are cached immediately after processing σ'_i. Hence, $X_{i,\ell} \subseteq Y_{i,\ell}$. By the definition of σ'_i, $|X_{i,\ell}|$ equals the number of color ℓ jobs in σ_i. Therefore, the number of color ℓ jobs in σ_i is at most $|Z_{i,\ell}|$. Since in each round, we execute as many jobs as the current reconfiguration allows in T', all color ℓ jobs in σ_i are executed in S in round i. Summing up over all colors ℓ and all rounds i, the lemma follows. □

Theorem 3.3. *Algorithm RLSerialize is resource competitive for rate-limited* $[\Delta \mid d_\ell \mid 1 \mid 1]$*, where* $\Delta < d_\ell$ *for all colors* ℓ*.*

Proof. Consider any instance (σ, m) of rate-limited $[\Delta \mid d_\ell \mid 1 \mid 1]$, where $\Delta < d_\ell$ for all colors ℓ. Suppose there exists an m-resource offline schedule T for σ with cost C. By Lemma 3.6.1, there exists an m-resource drop-free schedule for σ with cost at most $2C$. By Lemma 3.6.2, there exists a schedule for $\sigma' = \mathit{serializedReqSeq}(\sigma)$ with at most $\frac{4C}{\Delta}$ misses and a cache size of m. Let S' be the $3m$-resource schedule produced by *Mark* on (σ', m), and S be the $3m$-resource schedule produced by *RLSerialize* on (σ, m). Since σ is a request sequence for prefix paging, by Lemma 3.5.2, S' incurs cost $O(\frac{C}{\Delta})$. By Lemma 3.6.3, schedule S incurs cost $O(C)$. Hence, the theorem follows. □

3.7 Rate-Limited Batched Arrivals

In this section we solve rate-limited $[\Delta \mid d_\ell \mid D \mid D]$, which is characterized by a fixed reconfiguration cost Δ, per-color drop costs d_ℓ, a fixed delay bound D, batched arrivals (jobs arrive at integral multiples of D), and limited arrival rate (at most D jobs of the same color arrive at each integral multiple of D).

In this section, we reduce rate-limited $[\Delta \mid d_\ell \mid D \mid D]$ to two cases: (1) $\Delta \geq d_\ell$ for all colors ℓ, and (2) $\Delta < d_\ell$ for all colors ℓ. We solve the former case by a reduction to $[\Delta \mid d_\ell \mid 1 \mid 1]$, which is addressed in Section 3.4. The latter case is simpler. We solve the latter case by a reduction to a special case of $[\Delta \mid d_\ell \mid 1 \mid 1]$, referred to as rate-limited $[\Delta \mid d_\ell \mid 1 \mid 1]$, which is defined and solved in Section 3.6.

We refer to the portion of a request sequence that corresponds to a block as a *frame*. The rest of the section is organized as follows. In Section 3.7.1, we introduce some definitions. In Section 3.7.2, we present algorithm *Split*, the algorithm that schedules an entire request sequence. An important subroutine of algorithm *Split* is *Reshape*, which reshapes the entire request sequence by invoking *ReshapeFrame* (defined in Section 3.8.6), the algorithm that reshapes a frame. In Section 3.7.3, we show algorithm *Split* is resource competitive for $[\Delta \mid d_\ell \mid D \mid D]$; our proof uses Lemmas 3.8.17 and 3.8.18 of Section 3.8.6.

3.7.1 Definitions

In this section, we make use of the notion of pseudo-schedule, which is defined in Section 2.2. Consider an arbitrary instance (σ, m) of $[\Delta \mid d_\ell \mid D \mid D]$, and any pseudo-schedule P for σ. We define *adjReqSeq*(P) as a request sequence σ' such that (1) the set of jobs appearing in σ' is the same as that appearing in σ, (2) the arrival time of each job x in σ' is an arbitrary round if x is dropped in P, and otherwise the round in which x is executed in P, and (3) the delay bound of each job in σ' is set to 1. In this dissertation, we always let *adjReqSeq* take a pseudo-schedule that executes all jobs. We define *adjRlReqSeq*(P) in the same way as we define *adjReqSeq*(P), except that the set of jobs appearing in *adjRlReqSeq*(P) is the set of jobs executed in the first m resources in P. It is not hard to see that any schedule for *adjReqSeq*(P) (resp., *adjRlReqSeq*(P)) is also a schedule for σ.

Given an initial coloring μ and a pseudo-schedule P, we use *final*(P, μ) to denote the *final coloring* of P, that is, the coloring of P after the last round. For any two colorings μ and ν, we define the distance between μ and ν, denoted *dist*(μ, ν), as the number of resources that have distinct colors in μ and ν. We define $Cost(P, \mu, \nu)$ as $Cost(P, \mu) + \Delta \cdot dist(final(P, \mu), \nu)$. The preceding definitions turn out to be useful for analyzing pseudo-schedules and schedules, obtained via concatenation.

3.7.2 Algorithms *Reshape* and *Split*

Given an arbitrary instance (σ, m) of $[\Delta \mid d_\ell \mid D \mid D]$, algorithm *Reshape* generates a pseudo-schedule for σ. It independently decides the pseudo-schedule for each frame τ of σ by invoking *ReshapeFrame*(τ, μ_0) (defined in Section 3.8.6), where μ_0 is the default coloring. The final pseudo-schedule for σ is obtained by concatenating the pseudo-schedules for each frame i of σ, in increasing order of i.

Algorithm *Split* is defined as follows. We first consider two special cases. First we consider the special case in which each job appearing in σ has a drop cost at most Δ. In this case, algorithm *Split* proceeds in two stages. In the first stage, we obtain a pseudo-schedule P by applying algorithm *Reshape* on σ. In the second stage, we apply algorithm *Serialize* (defined in Section 3.4.1) on $adjReqSeq(P)$ to obtain a schedule for $adjReqSeq(P)$, which is also a schedule for σ.

Second we consider the special case in which each job appearing in σ has a drop cost greater than Δ. Algorithm *Split* proceeds similarly as in the first special case, except that we replace $adjReqSeq(P)$ with $adjRlReqSeq(P)$, and algorithm *Serialize* with *RLSerialize* (defined in Section 3.6.1) in this case.

In the general case, we break each request in σ into two requests, one consisting of the jobs with per-color drop costs at most Δ, and the other consisting of the jobs with per-color drop costs greater than Δ. Let α (resp., β) denote the resulting sequence of requests involving the jobs with per-color drop costs at most Δ (resp., greater than Δ). We double the number of resources, split the set of resources in half, and use the first half to execute the jobs in α as in the first special case, and the second half to execute the jobs in β as in the second special case.

3.7.3 Analysis of *Split*

Lemma 3.7.1. *Consider an arbitrary instance (σ, m) of $[\Delta \mid d_\ell \mid D \mid D]$ and any coloring μ. If there exists an m-resource schedule S for σ, then there exists an m-resource schedule S' for $adjReqSeq(Reshape(\sigma))$ such that $Cost(S') = O(Cost(S))$.*

Proof. For any nonnegative integer i, let σ_i denote frame i of σ, and let S_i denote the portion of S associated with σ_i. Let μ_0 be the default coloring. For any integer $i > 0$, we define μ_i as $\mathit{final}(S_i, \mu_{i-1})$. It is not hard to see that $\mathit{Cost}(S) = \mathit{Cost}(S, \mu_0) = \sum_i \mathit{Cost}(S_i, \mu_i)$.

By Lemma 3.8.17, for each nonnegative integer i, there exists an m-resource schedule S'_i for $\mathit{adjReqSeq}(\mathit{ReshapeFrame}(\sigma_i, \mu_0))$ such that

$$\mathit{Cost}(S'_i, \mu_i, \mu_{i+1}) = O(\mathit{Cost}(S_i, \mu_i)).$$

Let S' be the concatenation of the S'_i's, in increasing order of i. It is not hard to verify that S' is a schedule for $\mathit{adjReqSeq}(\mathit{Reshape}(\sigma))$, and that $\mathit{Cost}(S') = \mathit{Cost}(S', \mu_0) \leq \sum_i \mathit{Cost}(S'_i, \mu_i, \mu_{i+1})$. Hence, $\mathit{Cost}(S') = O(\mathit{Cost}(S))$. □

Lemma 3.7.2. *Consider an arbitrary instance (σ, m) of $[\Delta \mid d_\ell \mid D \mid D]$ and any coloring μ. If $\Delta < d_\ell$ for all colors ℓ, and there exists an m-resource schedule S for σ, then there exists an m-resource schedule S' for $\mathit{adjReqSeq}(\mathit{Reshape}(\sigma))$ such that $\mathit{Cost}(S') = O(\mathit{Cost}(S))$.*

Proof. Since $\Delta < d_\ell$ for all colors ℓ, it is not hard to see that for any frame τ of σ, $|\mathit{LtColors}(\tau)| = 0$. The remainder of the proof of this lemma is analogous to the proof of Lemma 3.7.1. We simply replace *adjReqSeq* with *adjRlReqSeq*, and Lemma 3.8.17 with Lemma 3.8.18. □

Lemma 3.7.3. *If $\Delta \geq d_\ell$ for all colors ℓ, then algorithm Split is resource competitive for $[\Delta \mid d_\ell \mid D \mid D]$.*

Proof. Consider an arbitrary instance (σ, m) for $[\Delta \mid d_\ell \mid D \mid D]$, where $\Delta \geq d_\ell$ for all colors ℓ. Let $\sigma' = \mathit{adjReqSeq}(\mathit{Reshape}(\sigma))$. Thus, by the definition of algorithm *Split*, the schedule $T = \mathit{Split}(\sigma)$ equals $\mathit{Serialize}(\sigma')$. Suppose there exists an m-resource offline schedule S for σ with cost C. By Lemma 3.7.1, there exists an m-resource schedule S' for σ' with cost $O(C)$. Since σ' is a request sequence for $[\Delta \mid d_\ell \mid 1 \mid 1]$, and by Theorem 3.2, algorithm *Serialize* is resource competitive for $[\Delta \mid d_\ell \mid 1 \mid 1]$, T incurs cost $O(C)$ with $O(m)$ resources. Hence, the lemma follows. $\square$

Lemma 3.7.4. *If $\Delta < d_\ell$ for all colors ℓ, algorithm Split is resource competitive for $[\Delta \mid d_\ell \mid D \mid D]$.*

Proof. The proof of this lemma is analogous to the proof of Lemma 3.7.3. We simply replace $\Delta \geq d_\ell$ with $\Delta < d_\ell$, $[\Delta \mid d_\ell \mid 1 \mid 1]$ with rate-limited $[\Delta \mid d_\ell \mid 1 \mid 1]$ (see Section 3.6), algorithm *Serialize* with *RLSerialize* (see Section 3.6.1), Lemma 3.7.1 with Lemma 3.7.2, and Theorem 3.2 with Theorem 3.3 in the proof of Lemma 3.7.3. $\square$

Theorem 3.4. *Algorithm Split is resource competitive for $[\Delta \mid d_\ell \mid D \mid D]$.*

Proof. Since algorithm *Split* reduces the general case to the two special cases in Lemmas 3.7.3 and 3.7.4, and uses disjoint set of resources to handle each special case, the theorem follows from Lemmas 3.7.3 and 3.7.4. $\square$

3.8 Rate-Limited Batched Arrivals: Reshaping a Frame

The purpose of this section is to provide the definition of algorithm *ReshapeFrame*, the algorithm that we use to reshape a frame, and to state and prove Lemmas 3.8.17 and 3.8.18. Algorithm *ReshapeFrame* is invoked in Section 3.7.2 to reshape and schedule an entire request sequence. Lemma 3.8.17 (resp., Lemma 3.8.18) is used in the proof of Lemma 3.7.1 (resp., Lemma 3.7.2) in Section 3.7.3.

We use algorithm *ReshapeFrame* to generate an assignment that assigns each job in a frame to a round, and to perform an offline to offline reduction. The remainder of this section is organized as follows. In Section 3.8.1, we give some preliminaries. In Sections 3.8.2 through 3.8.5, we introduce a set of functions and their properties that are useful to define and analyze algorithm *ReshapeFrame*. In Section 3.8.6, we define algorithm *ReshapeFrame*, and state and prove Lemma 3.8.17. One of the main lemmas that we use to prove Lemma 3.8.17, namely Lemma 3.8.16, bounds the cost of our offline to offline reduction. We prove Lemma 3.8.16 by considering two cases depending on the number of heavy colors in a frame. (The formal definition of a heavy color is given in Section 3.8.1.) Sections 3.8.7 and 3.8.8 handle the cases where there are many heavy colors and few heavy colors, respectively.

3.8.1 Preliminaries

Any pseudo-schedule (resp., schedule) mentioned in this section refers to a pseudo-schedule (resp., schedule) for a frame. Throughout this section, we use the integer m to denote the bound on the number of the resources that can be used by a schedule. For any integer i, we use $[i]$ to denote the sets of integers 0 through $i-1$. We define M as the set of resources $[m]$.

For any set X of jobs, we use $\mathit{seq}(X)$ to denote the sequence of jobs obtained by sorting the jobs in X in descending order of drop costs, breaking ties by color. For any sequence α of jobs, we use $\mathit{Set}(\alpha)$ to denote the set of jobs appearing in α. For any set of jobs X (resp., sequence of jobs α), we use $\mathit{Colors}(X)$ (resp., $\mathit{Colors}(\alpha)$) to denote the set of colors ℓ such that there exists a job in X of color ℓ.

Consider any frame τ. We define $\mathit{Jobs}(\tau)$ as the set of jobs appearing in τ. For any color ℓ, we define the *load* of ℓ, denoted $\mathit{load}(\tau,\ell)$, as the number of color ℓ jobs in $\mathit{Jobs}(\tau)$. We define the set of *heavy* colors, denoted $\mathit{HvyColors}(\tau)$, as the set of colors ℓ such that $\mathit{load}(\tau,\ell)\cdot d_\ell \geq \Delta$, and the set of *light* colors, denoted $\mathit{LtColors}(\tau)$, as the set of colors not in $\mathit{HvyColors}(\tau)$. We sort the colors in $\mathit{HvyColors}(\tau)$ in descending order of drop costs, breaking ties by load. We use $\mathit{PrmyHvyColors}(\tau)$ to denote the first $\min(m, |\mathit{HvyColors}(\tau)|)$ colors in $\mathit{HvyColors}(\tau)$, and $\mathit{SecHvyColors}(\tau)$ to denote the set of remaining colors in $\mathit{HvyColors}(\tau)$. We define $\mathit{PrmyHvyJobs}(\tau)$ (resp., $\mathit{SecHvyJobs}(\tau)$) as the set of jobs of the colors in $\mathit{PrmyHvyColors}(\tau)$ (resp., $\mathit{SecHvyColors}(\tau)$).

Consider any pseudo-schedule P. For any color ℓ, we define $\mathit{ExeSet}(P,\ell)$ (resp., $\mathit{DropSet}(P,\ell)$) as the set of color ℓ jobs that are executed (resp., dropped) in pseudo-schedule P. We define $\mathit{ExeSet}(P)$ (resp., $\mathit{DropSet}(P)$) as $\cup_\ell \mathit{ExeSet}(P,\ell)$ (resp., $\cup_\ell \mathit{DropSet}(P,\ell)$). We define $\mathit{exe}(P,i)$ as the sequence of the jobs that are executed on resource i in P. We define $\mathit{pre}(P,i)$ as the maximal monochromatic prefix of $\mathit{exe}(P,i)$, and $\mathit{suf}(P,i)$ as the suffix of $\mathit{exe}(P,i)$ obtained by removing $\mathit{pre}(P,i)$ from $\mathit{exe}(P,i)$. We define $\mathit{PreSet}(P)$ as $\cup_\ell \mathit{Set}(\mathit{pre}(S,i))$.

We define $\mathit{Full}(P)$ (resp., $\mathit{Empty}(P)$) as the set of resources i such that each slot in resource i is occupied (resp., free) in P. We define $\mathit{freeSlots}(P,i)$ as the sequence of slots in resource i that are free in P. For any pseudo-schedule P, we define a permutation π^P of the resources as follows. The resources i are ordered in ascending order of $|\mathit{pre}(P,i)|$, breaking ties by the color of the jobs in $\mathit{pre}(P,i)$ if possible, and arbitrarily otherwise (i.e., when $\mathit{pre}(P,i)$ is the empty sequence, or when there are ties in color). We define $\mathit{suf}(P)$ as the concatenation of the $\mathit{suf}(P,\pi^P(i))$'s, over all i in $[m]$, in increasing order of i. We define $\mathit{freeSlots}(P)$ as the concatenation of the $\mathit{freeSlots}(P,\pi^P(i))$'s, over all i in $[m]$, in increasing order of i.

Consider any frame τ, any pseudo-schedule P for τ, and any coloring μ. For any color ℓ, we use $\mathit{Rscs}(\mu,\ell)$ to denote the set of resources i in M such that $\mu(i)=\ell$. For any color ℓ, we define $\mathit{Mono}(P,\mu,\ell)$ as the set of resources i such that $\mu(i)=\nu(i)=\ell$, where $\nu=\mathit{final}(P,\mu)$, and all the jobs executed on resource i in P are color ℓ jobs. We define $\mathit{Mono}(P,\mu)$ as $\cup_\ell \mathit{Mono}(P,\mu,\ell)$. We define the resources used by P that are not in $\mathit{Mono}(P,\mu)$ as $\mathit{Multi}(P,\mu)$.

Consider any schedule S and any coloring μ. We define $\mathit{Mismatch}(S,\mu)$ as the set of resources i in M such that $|\mathit{pre}(S,i)|>0$ and the color of $\mathit{pre}(S,i)$ is different from $\mu(i)$.

We define function *extractedColoring* as follows. It takes a schedule S and a coloring μ, and returns a coloring ν constructed in the following manner: for any resource i, if resource i is in $Mono(S, \mu, \ell)$ for some color ℓ, then we set $\nu(i)$ to ℓ; otherwise, we set $\nu(i)$ to *black*.

Fact 3.8.1. *For any frame τ, any schedule S for τ, and any coloring μ, $Mono(S, \mu, \ell) = Rscs(\nu, \ell)$, where $\nu = extractedColoring(S, \mu)$.*

We define function *trunc* as follows. It takes a pseudo-schedule P and returns a schedule in which resources $[m]$ behave the same as in P.

Consider any schedule S. We say S is *prefix-complete* if for each resource i, $|pre(S, i)| > 0$. We say S is *suffix-free* if $|suf(S)| = 0$. We say S is *suffix-valuable* if for each resource i, the total drop cost of the jobs in $suf(S, i)$ is at least Δ. We say S is *light-free* if for any color ℓ in $LtColors(\tau)$, $ExeSet(S, \ell) = \emptyset$.

Consider two schedules S and T. We say S and T are *prefix-identical* if for each resource i, $pre(S, i) = pre(T, i)$, and *prefix-matched* if there exists a permutation π of the resources such that for each resource i, $pre(S, i) = pre(T, \pi(i))$.

Definition 3.8.1 (**Property Greedy-Packing).** We say a pair (X, S), where X is a set of jobs and S is a schedule, satisfies the Greedy-Packing property if $suf(S)$ is a prefix of $seq(X)$ of length $\min(|seq(X)|, |freeSlots(S')|)$, and is executed in the first s slots in $freeSlots(S')$, where S' is a schedule obtained by removing $suf(S)$ from S.

Consider any frame τ, any pseudo-schedule P for τ, and any coloring μ. We say a reconfiguration made by P is *external* if it is made in the first round of the block corresponding to τ, and *internal* otherwise. We use $\mathit{ExtReconfigCost}(P, \mu)$ to denote the cost incurred by the external reconfigurations made by P if given the initial coloring μ. We use $\mathit{IntReconfigCost}(P)$ to denote the cost incurred by internal reconfigurations made by P. For any schedule S, we define $\mathit{IntCost}(S)$ as the sum of $\mathit{IntReconfigCost}(S)$ and $\mathit{DropCost}(S)$.

Lemma 3.8.2. *For any frame τ, any schedule S for τ, and any coloring μ, $\mathit{Cost}(S, \mu) \geq |X| \cdot \Delta$, where X is the set of the colors ℓ in $\mathit{HvyColors}(\tau)$ such that $\mathit{Mono}(S, \mu, \ell) = \emptyset$.*

Proof. Consider any color ℓ in X. Since $X \subseteq \mathit{HvyColors}(\tau)$, by the definition of $\mathit{HvyColors}(\tau)$, $|\mathit{load}(\tau, \ell)| \cdot d_\ell \geq \Delta$. If $|\mathit{DropSet}(S, \ell)| = \mathit{load}(\tau, \ell)$, then the total drop cost of the jobs in $\mathit{DropSet}(S, \ell)$ is at least Δ. Otherwise, by the definition of X, any color ℓ job executed in S is executed on a resource in $\mathit{Multi}(S, \mu)$. Hence, in this case, there is at least one reconfiguration to color ℓ in S. Summing up over all colors ℓ in X, the lemma follows. $\square$

3.8.2 Function *prefixHvy*

The function *prefixHvy* takes a frame τ, a subset X of $\mathit{Jobs}(\tau)$ such that $|\mathit{Colors}(X)| \leq m$, and a coloring μ as arguments, and generates a schedule for τ. We implement *prefixHvy* by initializing a schedule S as a schedule for τ

that drops all jobs in $Jobs(\tau)$, and then modify S as follows. First, we define an injective f: $Colors(X)$ to $[m]$ that maximizes the number of colors ℓ in $Colors(X)$ such that $\mu(f(\ell)) = \ell$. Second, for each color ℓ in $Colors(X)$, we assign the jobs in X_ℓ to execute in the first $|X_\ell|$ slots on resource $f(\ell)$ in S, where X_ℓ is set of color ℓ jobs in X. Finally, we return S.

Fact 3.8.3. *Consider any frame τ, any coloring μ, and any subset X of $Jobs(\tau)$ such that $|Colors(X)| \leq m$. Let $S = prefixHvy(\tau, X, \mu)$. Then the following claims hold.*

1. *If $|Colors(X)| = m$, then schedule S is prefix-complete.*
2. *Schedule S is suffix-free.*
3. *$|Mismatch(S, \mu)| = k_\ell$, where k_ℓ is the number of colors ℓ in X such that $Rscs(\mu, \ell) = \emptyset$.*

Lemma 3.8.4. *Consider any frame τ, any schedule S, and any coloring μ. Let $\nu = extractedColoring(S, \mu)$. Let $T = prefixHvy(\tau, PrmyHvyColors(\tau), \nu))$. Then $Cost(S, \mu) = \Delta \cdot \Omega(|Mismatch(T, \nu)|)$.*

Proof. Let X (resp., Y) denote the set of colors ℓ in $HvyColors(\tau)$ (resp., $PrmyHvyColors(\tau)$) such that $Mono(S, \mu, \ell) = \emptyset$. Let Z denote the set of colors ℓ in $PrmyHvyColors(\tau)$ such that $Rscs(\nu, \ell) = \emptyset$. By (3) of Fact 3.8.3, $|Mismatch(T, \nu)| = |Z|$. By Fact 3.8.1, $Z = Y$. Since $PrmyHvyColors(\tau) \subseteq HvyColors(\tau)$, $Y \subseteq X$. The lemma then follows from Lemma 3.8.2. □

3.8.3 Function *pack*

The function *pack* takes a schedule S, and a subset X of $Jobs(\tau)$, where τ is the frame associated with S, as arguments, and generates a schedule for τ. We implement *pack* by initializing a schedule S' to S, and then modifying S' as follows. Let $\alpha = seq(X)$. We assign the prefix of α of length $s = \min(|\alpha|, |freeSlots(S)|)$ to the first s slots of $freeSlots(S)$. Finally we return schedule S'.

Fact 3.8.5. *Consider any schedule S for a frame τ, and a subset X of $Jobs(\tau)$. Let $T = pack(S, X)$.*

1. *If $X = \emptyset$, then $T = S$.*
2. *If S is prefix-complete and suffix-free, then (X, T) and $(Set(suf(T)), T)$ each satisfy the Greedy-Packing property.*
3. *If S is prefix-complete, then schedules S and T are prefix-identical.*

3.8.4 Function *dropSuf*

The function *dropSuf* takes a schedule S for a frame, and generates another schedule for the same frame. We implement *dropSuf* by initializing a schedule S' to S, and then modifying S' in the following manner: For each resource i, if the total drop cost of the jobs in $suf(S', i)$ is less than Δ, we drop the jobs in $suf(S', i)$ from resource i in S'. Finally we return schedule S'.

Fact 3.8.6. *Consider any schedule S. Let schedule $T = \mathit{dropSuf}(S)$. Then the following claims hold.*

1. *Schedule T is suffix-valuable.*

2. *If $(\mathit{Set}(\mathit{suf}(S)), S)$ satisfies the Greedy-Packing property, then $\mathit{suf}(T)$ is a prefix of $\mathit{suf}(S)$, and $(\mathit{Set}(\mathit{suf}(T)), T)$ satisfies the Greedy-Packing property.*

3. *If T is suffix-free, then $T = S$.*

4. *Schedules S and T are prefix-identical.*

5. *$\mathit{IntCost}(T) \leq \mathit{IntCost}(S)$.*

Fact 3.8.7. *Consider any pair of schedules (S, T), and any two sets of jobs X and Y such that $X \subseteq Y$. Let $S' = \mathit{dropSuf}(S)$ and $T' = \mathit{dropSuf}(T)$. If (1) S and T are prefix-matched, (2) (X, S) satisfies the Greedy-Packing property, and (3) (Y, T) satisfies Greedy-Packing, then the following claims hold.*

1. *$|\mathit{suf}(S')| \leq |\mathit{suf}(T')|$, and the total drop cost of the jobs in $\mathit{suf}(S')$ is at most that of the jobs in $\mathit{suf}(T')$.*

2. *$\mathit{DropCost}(S') \geq \Delta \cdot (|\mathit{hasSuf}(T')| - |\mathit{hasSuf}(S')|)$.*

3.8.5 Function *prefixLight*

The function *prefixLight* takes a schedule S for a frame, and a coloring μ as arguments, and returns a pseudo-schedule for the same frame. We implement *prefixLight* by initializing P to S, and then modify P iteratively. Each iteration proceeds as follows. If $DropSet(P) = \emptyset$, then we return P and terminate the procedure. Otherwise, we proceed in the following stages. In the first stage, we pick any color ℓ such that $DropSet(P, \ell) \neq \emptyset$. In the second stage, if there exists a resource i in $Empty(P) \cap M$ such that in $\mu(i) = \ell$, we pick an arbitrary such resource i; otherwise, we pick a resource i in $Empty(P) \setminus M$ with the smallest index. In the third stage, we modify P by assigning all jobs in $DropSet(P, \ell)$ to execute in the first $|DropSet(P, \ell)|$ slots in resource i. Finally we return pseudo-schedule P.

Fact 3.8.8. *Consider any schedule S for a frame and any coloring μ. Let $T = trunc(prefixLight(S, \mu))$. Then the following claims hold.*

1. *For each resource i, $suf(T, i) = suf(S, i)$.*
2. *If S is prefix-complete, then $T = S$.*
3. *$Mismatch(T, \mu) = Mismatch(S, \mu)$.*
4. *For each color ℓ such that $DropSet(T, \ell) \neq \emptyset$, there does not exist resource i in $M \setminus Mismatch(T, \mu)$ such that $\mu(i) = \ell$.*

3.8.6 Algorithm *ReshapeFrame*

Algorithm *ReshapeFrame* takes a frame τ and a coloring μ as arguments, and generates a pseudo-schedule for τ. Let X be the set of jobs of the colors in $PrmyHvyColors(\tau)$, and Y be the set of jobs of colors in $SecHvyColors(\tau)$. We implement *ReshapeFrame* in the following four phases.

In the first phase, we set $S_1 = \mathit{prefixHvy}(\tau, X, \mu)$. Note that $\mathit{Colors}(X) \leq m$. In the second phase, we set $S_2 = \mathit{pack}(S_1, Y)$. In the third phase, we set $S_3 = \mathit{dropSuf}(S_2)$. In the fourth phase, we set $P = \mathit{prefixLight}(S_3, \mu)$, and then return P.

Fact 3.8.9. *Consider any frame τ, and any two colorings μ and ν. Let $P = \mathit{ReshapeFrame}(\tau, \mu)$ and $Q = \mathit{ReshapeFrame}(\tau, \nu)$. Then there exists a function $f\colon M \times N$, where N is the set of resources used by P, such that for each resource i in M, the sequence of jobs executed on resource i in Q equals the sequence of jobs executed on resource $f(i)$ in P.*

Fact 3.8.10. *Consider any frame τ, and any two colorings μ and ν. Let $P = \mathit{ReshapeFrame}(\tau, \mu)$ and $Q = \mathit{ReshapeFrame}(\tau, \nu)$. If $|\mathit{LtColors}(\tau)| = 0$, then there exists a function $f\colon M \times M$ such that for each resource i in M, the sequence of jobs executed on resource i in Q equals the sequence of jobs executed on resource $f(i)$ in P.*

Lemma 3.8.11. *Consider any frame τ, and any two colorings μ and ν. Let $P = \mathit{ReshapeFrame}(\tau, \mu)$, $Q = \mathit{ReshapeFrame}(\tau, \nu)$, and $T = \mathit{trunc}(Q)$. Then T is a schedule for $\mathit{adjReqSeq}(P)$.*

Proof. By definition of $\mathit{adjReqSeq}(P)$, all jobs in $\mathit{adjReqSeq}(P)$ have delay bound 1. Hence, we only need to show that, for each round j and each color ℓ, the total number of color ℓ jobs executed in T is at most the total number of color ℓ jobs arriving in round j in $\mathit{adjReqSeq}(P)$. The lemma then follows from Fact 3.8.9. $\square$

Lemma 3.8.12. *Consider any frame τ, and any two colorings μ and ν. Let $P = \mathit{ReshapeFrame}(\tau, \mu)$, $Q = \mathit{ReshapeFrame}(\tau, \nu)$, and $T = \mathit{trunc}(Q)$. If $|\mathit{LtColors}(\tau)| = 0$, then T is a schedule for $\mathit{adjRlReqSeq}(P)$.*

Proof. The proof of this lemma is analogous to the proof of Lemma 3.8.11. We simply replace $\mathit{adjReqSeq}(P)$ with $\mathit{adjRlReqSeq}(P)$, and Fact 3.8.9 with Fact 3.8.10. $\square$

Lemma 3.8.13. *Consider any frame τ, any schedule S for τ, and any coloring μ. Let $T = \mathit{trunc}(\mathit{ReshapeFrame}(\tau, \nu))$, where $\nu = \mathit{extractedColoring}(S, \mu)$. If $|\mathit{HvyColors}(\tau)| > m$, then $\mathit{Cost}(T, \mu) = O(\mathit{Cost}(S, \mu))$.*

Proof. See Section 3.8.7. $\square$

Lemma 3.8.14. *Consider any frame τ, any schedule S for τ, and any coloring μ. Let $T = \mathit{trunc}(\mathit{ReshapeFrame}(\tau, \nu))$, where $\nu = \mathit{extractedColoring}(S, \mu)$. If $|\mathit{HvyColors}(\tau)| \leq m$, then $\mathit{Cost}(T, \mu) = O(\mathit{Cost}(S, \mu))$.*

Proof. See Section 3.8.8. $\square$

Lemma 3.8.15. *For any frame τ, any two schedules S and T for τ, and any coloring μ, $\mathit{dist}(\mathit{final}(T, \mu), \mathit{final}(S, \mu)) \cdot \Delta \leq \mathit{Cost}(S, \mu) + \mathit{Cost}(T, \mu)$.*

Proof. For any resource i in $\mathit{Mono}(T, \mu) \cap \mathit{Mono}(S, \mu)$, $\mu'(i) = \mu''(i)$, where $\mu' = \mathit{final}(T, \mu)$ and $\mu''(i) = \mathit{final}(S, \mu)$. Hence, $\mathit{dist}(\mathit{final}(T, \mu), \mathit{final}(S, \mu)) \leq |\mathit{Multi}(T, \mu) \cup \mathit{Multi}(S, \mu)|$, and the lemma follows. $\square$

Lemma 3.8.16. *Consider any frame τ, any schedule S for τ, and any coloring μ. Let $\nu = \mathit{extractedColoring}(S, \mu)$, and $T = \mathit{trunc}(\mathit{ReshapeFrame}(\tau, \nu))$. Then $\mathit{Cost}(T, \mu, \mathit{final}(S, \mu)) = O(\mathit{Cost}(S, \mu))$.*

Proof. The claim follows from Lemmas 3.8.13, 3.8.14, and 3.8.15. □

Lemma 3.8.17. *For any frame τ and any coloring μ', if there exists an m-resource schedule S for τ, then there exists an m-resource schedule T for $\mathit{adjReqSeq}(\mathit{ReshapeFrame}(\tau, \mu'))$ such that for any coloring μ,*

$$\mathit{Cost}(T, \mu, \mathit{final}(S, \mu)) = O(\mathit{Cost}(S, \mu)).$$

Proof. Let $\nu = \mathit{extractedColoring}(S, \mu)$ and $T = \mathit{trunc}(\mathit{ReshapeFrame}(\tau, \nu))$. By Lemma 3.8.11, T is a schedule for $\mathit{adjReqSeq}(\mathit{ReshapeFrame}(\tau, \mu'))$. By Lemma 3.8.16, $\mathit{Cost}(T, \mu, \mathit{final}(S, \mu)) = O(\mathit{Cost}(S, \mu))$. □

Lemma 3.8.18. *For any frame τ and any coloring μ', if $|\mathit{LtColors}(\tau)| = 0$, and if there exists an m-resource schedule S for τ, then there exists an m-resource schedule T for $\mathit{adjRlReqSeq}(\mathit{ReshapeFrame}(\tau, \mu'))$ such that for any coloring μ,*

$$\mathit{Cost}(T, \mu, \mathit{final}(S, \mu)) = O(\mathit{Cost}(S, \mu)).$$

Proof. The proof of this lemma is analogous to the proof of Lemma 3.8.17; we obtain the proof of this lemma by replacing *adjReqSeq* with *adjRlReqSeq*, and Lemma 3.8.11 with Lemma 3.8.12 in the proof of Lemma 3.8.17. □

3.8.7 Many Heavy Colors

The purpose of this section is to provide the proof of Lemma 3.8.13 (stated in Section 3.8.6). The organization of this section is as follows. Section 3.8.7 gives some preliminaries. Sections 3.8.7.2 through 3.8.7.4 introduce a set of functions that are useful to prove Lemma 3.8.13. Section 3.8.7.5 establishes a set of useful lemmas, and then obtains the proof of Lemma 3.8.13.

3.8.7.1 Preliminaries

This section provides some useful definitions, facts, and lemmas used for the analysis. We say that a schedule S is *prefix-multichromatic* if, for any two resources i and j such that $i \neq j$ and $|pre(S,i)|$, $|pre(S,j)| > 0$, the color of $pre(S,i)$ is different from that of $pre(S,j)$. We say that a schedule S is *prefix-primary-heavy* if S is prefix-multichromatic, and $PreSet(S) = PrmyHvyJobs(\tau)$, where τ is the frame associated with S. We say that a schedule S is *prefix-dominant* if S is prefix-multichromatic, and $PreSet(S)$ equals the set of jobs of the $\min(|Colors(ExeSet(S))|, m)$ colors in $Colors(ExeSet(S))$ with the highest drop costs.

Fact 3.8.19. *For any frame τ and any two prefix-primary-heavy schedules S and T for τ, S and T are prefix-matched.*

Fact 3.8.20. *For any frame τ, if $|HvyColors(\tau)| > m$, then any prefix-primary-heavy schedule S for τ is prefix-complete.*

Lemma 3.8.21. *Consider any frame τ and any schedule S for τ. Let k be the* *number of light colors ℓ such that $ExeSet(S, \ell) \neq \emptyset$. If $|HvyColors(\tau)| > m$, then $IntCost(S) \geq \Delta \cdot (|HvyColors(\tau)| + k - m)$.*

Proof. Suppose there are p, $0 \leq p \leq |HvyColors(\tau)|$, colors in $HvyColors(\tau)$ such that $ExeSet(S, \ell) = \emptyset$. By the definition of $HvyColors(\tau)$, $DropCost(S) \geq p \cdot \Delta$. Let $q = |HvyColors(\tau)| - p$. Thus, there are $q + k$ colors ℓ such that $ExeSet(S, \ell) \neq \emptyset$. Hence, $IntReconfigCost(S) \geq (q + k - m) \cdot \Delta$. Hence, the lemma follows. $\square$

We define $hasSuf(P)$ as the set of resources i such that $suf(P, i) \neq \emptyset$.

Lemma 3.8.22. *For any frame τ and any schedule S for τ, if $(Set(suf(S)), S)$ satisfies the Greedy-Packing property, then*

$$IntReconfigCost(S) = \Delta \cdot \Theta(|Colors(suf(S))| + |hasSuf(S)|).$$

Proof. Since $(Set(suf(S)), S)$ satisfies the Greedy-Packing property, an internal reconfiguration on a resource i in S is either a reconfiguration from $pre(S, i)$ to $suf(S, i)$, or from a color ℓ to a distinct color ℓ' in $Colors(suf(S))$. Hence the lemma follows. $\square$

3.8.7.2 Function *dropLights*

The function *dropLights* takes a schedule S for a frame, and returns a schedule the same frame. We implement *dropLights* by initializing a schedule T to S, and then modifying T as follows. Let τ be the frame associated with S. For any color ℓ in $LtColors(\tau)$, we drop color ℓ jobs from T. Finally we return T.

Lemma 3.8.23. *Consider any frame τ and any schedule S for τ. Let $T =$ dropLights(S). If $|HvyColors(\tau)| > m$, then for any coloring μ, $Cost(T, \mu) = O(Cost(S, \mu)$.*

Proof. It is not hard to see that $ReconfigCost(T, \mu) \leq ReconfigCost(S, \mu)$. Let k be the number of colors in $LtColors(\tau)$ such that $ExeSet(T, \ell) \neq \emptyset$. By the definition of $LtColors(\tau)$, $load(\ell) \cdot d_\ell < \Delta$. Hence, $DropCost(T) \leq DropCost(S) + k \cdot \Delta$. Since $|HvyColors(\tau)| > m$, Lemma 3.8.21 implies that $IntCost(S) \geq k \cdot \Delta$. Hence, the lemma follows. □

Fact 3.8.24. *For any frame τ and any schedule S for τ, dropLights(S) is a light-free schedule for τ.*

3.8.7.3 Function *canonicalize*

Given a schedule S for a frame and a coloring μ, function *canonicalize* returns a schedule for the same frame. In the following, we describe an implementation of function *canonicalize*. Let X be the subset of $Colors(ExeSet(S))$ that consists of the $\min(|Colors(ExeSet(S))|, m)$ colors with the highest per-color drop costs, breaking ties by load. Let Y be the set of jobs of the colors in X. Let $Z = ExeSet(S) \setminus Y$. Let τ be the frame associated with S. We define $T' = prefixHvy(\tau, Y, \mu)$ (note that $|Colors(Y)| = |X| \leq m$) and $T = pack(T', Z)$, and then return schedule T.

We define function *canonicalize*$'$ (resp., *canonicalize*$''$) in the same way as we define function *canonicalize* except that the set X in *canonicalize*$'$

(resp., *canonicalize*$''$) is the subset of $\mathit{Colors}(\mathit{ExeSet}(S))$ that consists of the $\min(|\mathit{Colors}(\mathit{ExeSet}(S))|, m)$ colors of the highest (resp., lowest) load, breaking ties arbitrarily.

Fact 3.8.25. *Consider any frame τ, any schedule S for τ, and any coloring μ. Let $T = \mathit{canonicalize}(S, \mu)$.*

1. *The set $\mathit{ExeSet}(T)$ equals $\mathit{ExeSet}(S)$.*
2. *If S is light-free, then T is light-free.*
3. *The schedule T is prefix-dominant.*
4. *If S is light-free, then $\mathit{Set}(\mathit{suf}(T)) \subseteq \mathit{SecHvyJobs}(\tau)$.*
5. *The pair $(\mathit{Set}(\mathit{suf}(T)), T)$ satisfies the Greedy-Packing property.*
6. *If S is light-free and prefix-primary-heavy, then T is a prefix-primary-heavy schedule for τ.*

Lemma 3.8.26. *Consider any frame τ, any schedule S for τ, and any coloring μ. Let S' be any schedule for τ that executes the same set of jobs as S. Let $S'' = \mathit{canonicalize}'(S, \mu)$. Then $|\mathit{hasSuf}(S'')| \le |\mathit{hasSuf}(S')|$.*

Proof. Let $h' = |\mathit{hasSuf}(S')|$ and $h'' = |\mathit{hasSuf}(S'')|$. We prove the lemma by contradiction. Suppose $h'' > h'$. Let $X' = \mathit{hasSuf}(S')$ and $Y' = M \setminus X'$. Let X'' be the set of resources $\pi^{S''}(0)$ through $\pi^{S''}(h' - 1)$. (The permutation π^S is defined in Section 3.8.1.) Let $Y'' = M \setminus X''$.

By the definition of *canonicalize*$'$, since $h' < h''$, the total number of jobs executed in X'' in S'' equals $h' \cdot D$, which is at least the total number of jobs that can be executed in X' in S'. By the definition of *canonicalize*$'$, since $h' < h''$, the total number of jobs executed in Y'' in S'' is greater than the total load of the $(m - h'')$ colors in $Colors(ExeSet(S))$ with the largest load, which is greater than the total number of jobs executed in Y' in S'. This indicates that the total number of jobs executed in S'' is greater than the total number of jobs in S', contradiction. Hence, the assumption does not hold and the lemma follows. $\square$

With a similar proof as that of Lemma 3.8.26, we obtain the following lemma.

Lemma 3.8.27. *Consider any frame τ, any schedule S for τ, and any coloring μ. Let $S' = canonicalize(S, \mu)$. Let $S'' = canonicalize''(S, \mu)$. Then $|hasSuf(S'')| \geq |hasSuf(S')|$.*

Lemma 3.8.28. *Consider any frame τ, and any schedule S for τ. Let μ be any coloring. Let $S' = canonicalize'(S, \mu)$, and $S'' = canonicalize''(S, \mu)$. If $|HvyColors(S)| = m + k$ for some integer $k > 0$, then*

$$|hasSuf(S'')| - |hasSuf(S')| \leq k.$$

Proof. Let $h = |hasSuf(S')|$. By the definition of *canonicalize*$''$, we only need to show the the following claim: The total load of the $k + h$ colors in $Colors(ExeSet(S))$ of the smallest load, and the k colors in $Colors(ExeSet(S))$ of the largest load, is at most $(k + h) \cdot D$. By the definition of *canonicalize*$'$, it is not hard to see that the total load of the $k+h$ colors in $\mathit{Colors}(\mathit{ExeSet}(S))$ with the smallest load is at most $h \cdot D$. Since for each color ℓ, at most D jobs arrive in a frame, the total load of any k colors in $\mathit{Colors}(\mathit{ExeSet}(S))$ is at most $k \cdot D$. Hence, the lemma follows. $\square$

Lemma 3.8.29. *Consider any frame τ, any schedule S for τ, and any coloring μ. Let $T = \mathit{canonicalize}(S, \mu)$. If $|\mathit{HvyColors}(\tau)| > m$, then $\mathit{IntCost}(T) = O(\mathit{IntCost}(S))$.*

Proof. (1) of Fact 3.8.25 implies $\mathit{DropCost}(T) = \mathit{DropCost}(S)$. By (5) of Fact 3.8.25 and Lemma 3.8.22, we have $\mathit{IntCost}(T) = \Delta \cdot \Theta(|\mathit{Colors}(\mathit{suf}(S))| + |\mathit{hasSuf}(S)|)$. By the definition of *canonicalize*, it is not hard to see that

$$|\mathit{Colors}(\mathit{suf}(S))| = \max(|\mathit{Colors}(\mathit{ExeSet}(S))| - m, 0).$$

By Lemma 3.8.21, we have $|\mathit{Colors}(\mathit{suf}(S))| \leq \mathit{IntCost}(S)$. It remains to bound $|\mathit{hasSuf}(T)|$.

Let $k = |\mathit{HvyColors}(\tau)| - m$. Let $S' = \mathit{canonicalize}'(S, \mu)$ and $S'' = \mathit{canonicalize}''(S, \mu)$. By Lemma 3.8.26, we have $|\mathit{hasSuf}(S')| \leq |\mathit{hasSuf}(S)|$. By Lemma 3.8.27, $|\mathit{hasSuf}(T)| \leq |S''|$. By Lemma 3.8.28, $|\mathit{hasSuf}(S'')| \leq |\mathit{hasSuf}(S')| + k$. Hence, $|\mathit{hasSuf}(T)| \leq |\mathit{hasSuf}(S'')| \leq |\mathit{hasSuf}(S')| + k \leq |\mathit{hasSuf}(S)| + k$. By Lemma 3.8.21, $|\mathit{hasSuf}(T)| \cdot \Delta = O(\mathit{IntCost}(S))$. $\square$

3.8.7.4 Function *matchPrefix*

The function *matchPrefix* takes two schedules S and T for the same frame, and returns a schedule. We implement *matchPrefix* by initializing S'

to S, and then modifying S' as follows. First, we define a permutation π of the resources that maximizes the number of resources i such that the color of $pre(T, i)$ equals that of $pre(S, \pi(i))$. Second, for each resource i, we modify S' by executing all jobs in $pre(T, i)$ in the first $s = |pre(T, i)|$ slots on resource $\pi(i)$ in S'. Finally, we return S'.

Lemma 3.8.30. *Consider any frame τ, any schedule S for τ, and any coloring μ. Let T be any prefix-primary-heavy schedule for τ, and Let $S' = matchPrefix(S, T)$. If $HvyColors(\tau) > m$, and S is light-free and prefix-dominant, then the following claims hold.*

1. *Schedule S' is a prefix-primary-heavy schedule for τ.*

2. *$Set(suf(S')) \subseteq SecHvyJobs(\tau)$.*

3. *Schedule S' is light-free.*

4. *$IntCost(S') = O(IntCost(S))$.*

Proof. Since S is prefix-dominant, for each color ℓ in $PrmyHvyColors(\tau)$ such that $ExeSet(S, \ell) \neq \emptyset$, there exists a resource i such that $Set(pre(S, i)) = ExeSet(S, \ell)$.

From the above property of S, and the assumption that T is a prefix-primary-heavy schedule for τ, the permutation π of the resources in the definition of $matchPrefix(S, T)$ satisfies the following condition: For each resource i, if $|pre(T, i)| > 0$, then either (a) $ExeSet(S, \ell) = \emptyset$, or (b) $Set(pre(S, \pi(i))) = ExeSet(S, \ell)$ and $|pre(S, \pi(i))| \leq |pre(T, i)|$, where ℓ is the color of $pre(T, i)$.

By the definition of function *matchPrefix* and the property of the permutation π, it is not hard to see that S' is a schedule for τ. From the property of the permutation and the assumption that S is light-free, (2) and (3) follow, and $\mathit{DropCost}(S') \leq \mathit{DropCost}(S)$.

By the definition of function *matchPrefix*, for each resource i such that $|\mathit{pre}(T, i)| > 0$, we have $\mathit{pre}(S', \pi(i)) = \mathit{pre}(T, i)$. Since $|\mathit{HvyColors}(\tau)| > m$ and T is a prefix-primary-heavy schedule, by Fact 3.8.20, T is prefix-complete. Hence, S' and T are prefix-matched. Since S' is a schedule for τ, and T is a prefix-primary-heavy schedule for τ, (1) follows.

It remains to bound $\mathit{IntReconfigCost}(S')$. Consider any resource i. We have argued above that $|\mathit{pre}(T, i)| > 0$. Let ℓ be the color of $\mathit{pre}(T, i)$. We first consider the case where $\mathit{ExeSet}(S, \ell) \neq \emptyset$. By the aforementioned property of the permutation π, the color of $\mathit{pre}(S, \pi(i))$ equals the color of $\mathit{pre}(S, i)$, and the number of internal reconfigurations of resource i in S' equals that associated with resource $\pi(i)$ in S. We then consider the case where $\mathit{ExeSet}(S, \ell) = \emptyset$. In this case, the number of internal reconfigurations of resource i in S' is at most one greater than that associated with resource $\pi(i)$ in S. Since T is prefix-primary-heavy, $\ell \in \mathit{HvyColors}(\tau)$. By definition of $\mathit{HvyColors}(\tau)$, the total drop cost of the jobs in $\mathit{ExeSet}(S, \ell)$ is at least Δ. Summing over all resources i, $\mathit{IntReconfigCost}(S') \leq \mathit{IntReconfigCost}(S) + \mathit{DropCost}(S)$. Hence, (4) follows. $\square$

3.8.7.5 Cost Analysis

Lemma 3.8.31. *Consider any frame τ, any schedule S for τ, and any coloring μ. Let T be any prefix-primary-heavy schedule. Let $S_1 = dropLights(S)$, $S_2 = canonicalize(S_1, \mu)$, $S_3 = matchPrefix(S_2, T)$, $S_4 = canonicalize(S_3, \mu)$, and $S_5 = dropSuf(S_4)$. If $|HvyColors(\tau)| > m$, then the following claims hold.*

1. *Schedules S_4 and S_5 are prefix-primary-heavy schedules for τ.*

2. *$Set(suf(S_4)) \subseteq SecHvyJobs(\tau)$.*

3. *Each of the pairs $(Set(suf(S_4)), S_4)$ and $(Set(suf(S_5)), S_5)$ satisfies the Greedy-Packing property.*

4. *$IntCost(S_5) = O(Cost(S, \mu))$.*

Proof. For convenience of proof, we first establish a set of properties of each of the schedules S_1 through S_5, and then establish the claims in the lemma using the established properties.

We establish the following properties of S_1: (a) S_1 is light-free; (b) $Cost(S_1, \mu) = O(Cost(S, \mu))$. Property (a) of S_1 follows from Fact 3.8.24. Property (b) of S_1 follows from Lemma 3.8.23.

We establish the following properties of S_2: (a) S_2 is light-free; (b) S_2 is prefix-dominant; (c) $Set(suf(S_2)) \subseteq SecHvyJobs(\tau)$; (d) $IntCost(S_2) = O(IntCost(S_1))$. Property (a) of S_2 follows from property (a) of S_1, and (2) of Fact 3.8.25. Property (b) of S_2 follows from (3) of Fact 3.8.25. Property (c) of S_2 follows from property (a) of S_1, and (4) of Fact 3.8.25. Property (d) of S_2 follows from Lemma 3.8.29 and the assumption that $\mathit{HvyColors}(\tau) > m$.

We establish the following properties of S_3: (a) S_3 is a prefix-primary-heavy schedule for τ; (b) $\mathit{Set}(\mathit{suf}(S_3)) \subseteq \mathit{SecHvyJobs}(\tau)$; (c) S_3 is light-free; (d) $\mathit{IntCost}(S_3) = O(\mathit{IntCost}(S_2))$. Properties (a) through (d) of S_3 follows from the fact that T is a prefix-primary-heavy schedule for τ, Properties (a) and (b) of S_2, and Lemma 3.8.30.

We establish the following properties of S_4: (a) S_4 is a prefix-primary-heavy schedule for τ; (b) $\mathit{Set}(\mathit{suf}(S_4)) \subseteq \mathit{SecHvyJobs}(\tau)$; (c) $(\mathit{Set}(\mathit{suf}(S_4)), S_4)$ satisfies the Greedy-Packing property. (d) $\mathit{IntCost}(S_4) = O(\mathit{IntCost}(S_3))$. Properties (a) through (c) of S_4 follow from properties (a) and (c) of S_3, and Fact 3.8.25. Property (d) of S_4 follows from Lemma 3.8.29 and the assumption that $\mathit{HvyColors}(\tau) > m$.

We establish the following properties of S_5: (a) S_5 is a prefix-primary-heavy schedule for τ; (b) $\mathit{IntCost}(S_5) \leq \mathit{IntCost}(S_4)$; (c) $(\mathit{Set}(\mathit{suf}(S_5)), S_5)$ satisfies the Greedy-Packing property. Property (a) of S_5 follows from property (a) of S_4 and (4) of Fact 3.8.6. Property (b) of S_5 follows from (5) of Fact 3.8.6. Property (c) of S_5 follows from property (c) of S_4 and (2) of Fact 3.8.6.

Now we established the claims in the lemma. Claim (1) follows from property (a) of S_4 and property (b) of S_5. Claim (2) follows from property (b) of S_4. Claim (3) follows from property (c) of S_4 and property (c) of S_5. Claim (4) follows from property (b) of S_1, property (d) of S_2, property (d) of

S_3, property (d) of S_4, and property (b) of S_5. $\square$

Lemma 3.8.32. *For any frame τ, let $T_1 = \mathit{prefixHvy}(\tau, \mathit{PrmyHvyJobs}(\tau), \mu)$, where μ is any coloring, $T_2 = \mathit{pack}(T_1, \mathit{SecHvyJobs}(\tau))$, and $T_3 = \mathit{dropSuf}(T_2)$. If $|\mathit{HvyColors}(\tau)| > m$, then the following claims hold.*

1. *Schedules T_2 and T_3 are prefix-primary-heavy schedules for τ.*
2. *Any pairs of the schedules T_1 through T_3 are prefix-identical.*
3. *The sequence $\mathit{suf}(T_3)$ is a prefix of $\mathit{SecHvyJobs}(\tau)$.*
4. *The pairs $(\mathit{SecHvyJobs}(\tau), T_2)$, $(\mathit{Set}(\mathit{suf}(T_2)), T_2)$, and $(\mathit{Set}(\mathit{suf}(T_3)), T_3)$ each satisfy the Greedy-Packing property.*
5. *Schedule T_3 equals $\mathit{trunc}(\mathit{ReshapeFrame}(\tau, \mu))$.*

Proof. We first establish some properties of each of the schedules T_1 through T_3, and then establish the claims in the lemma using the established properties.

We establish the following properties of T_1: (a) schedule T_1 is a prefix-primary-heavy schedule for τ; (b) schedule T_1 is prefix-complete; (c) schedule T_1 is suffix-free. Property (a) of T_1 follows from the definition of function *prefixHvy*. Property (b) of T_1 follows from (1) of Fact 3.8.3, and the assumption that $|\mathit{HvyColors}(\tau)| > m$. Property (c) T_1 follows from (2) of Fact 3.8.3.

We establish the following properties of T_2: (a) schedule T_2 is a prefix-primary-heavy schedule for τ; (b) schedules T_2 and T_1 are prefix-identical, and T_2 is prefix-complete; (c) the pairs $(\mathit{SecHvyJobs}(\tau), T_2)$ and $(\mathit{Set}(\mathit{suf}(T_2)), T_2)$

satisfy Property Greedy-Packing, respectively. Properties (a) and (b) of T_2 follow from properties (a) and (b) of T_1, and (3) of Fact 3.8.5. Property (c) of T_3 follows from properties (b) and (c) of S_1, and (2) of Fact 3.8.5.

We establish the following properties of T_3: (a) schedule T_3 is a prefix-primary-heavy schedule for τ; (b) schedules T_3 and T_2 are prefix-identical, and T_3 is prefix-complete; (c) the pair $(\mathit{Set}(\mathit{suf}(T_3)), T_3)$ satisfies the Greedy-Packing property. (d) the sequence $\mathit{suf}(T_3)$ is a prefix of $\mathit{seq}(\mathit{SecHvyJobs}(\tau))$. Properties (a) and (b) of T_3 follow from properties (a) and (b) of T_2, and (4) of Fact 3.8.6. Property (c) of T_3 follows from property (c) of T_2, and (2) of Fact 3.8.6. From property (c) of T_2 implies that $\mathit{suf}(T_2)$ is a prefix of $\mathit{SecHvyJobs}(\tau)$. Property (d) of T_3 then follows from (2) of Fact 3.8.6.

We now establish the claims in the lemma. Claim (1) follows from property (a) of T_2 and property (a) of T_3. Claim (2) follows from property (b) of T_2 and property (b) of T_3. Claim (3) is property (d) of T_3. Claim (4) follows from property (c) of T_2 and property (c) of T_3. Claim (5) follow from property (b) of T_3, Claim (2) of Fact 3.8.8, and the definition of *ReshapeFrame*. □

Lemma 3.8.33. *Consider any frame τ, any schedule S for τ, and any two colorings μ and ν. Let $T = \mathit{trunc}(\mathit{ReshapeFrame}(\tau, \nu))$. If $|\mathit{HvyColors}(\tau)| > m$, then $\mathit{IntCost}(T) = O(\mathit{Cost}(S, \mu))$.*

Proof. For convenience of proof, we first define a set of schedules as follows. Let $T_1 = \mathit{prefixHvy}(\tau, \mathit{PrmyHvyJobs}(\tau), \nu)$, $T_2 = \mathit{pack}(T_1, \mathit{SecHvyJobs}(\tau))$, and $T_3 = \mathit{dropSuf}(T_2)$. Let $S_1 = \mathit{dropLights}(S)$, $S_2 = \mathit{canonicalize}(S_1, \mu)$, $S_3 =$

$\mathit{matchPrefix}(S_2, T_2)$, $S_4 = \mathit{canonicalize}(S_3, \mu)$, and $S_5 = \mathit{dropSuf}(S_4)$. Since $|\mathit{HvyColors}(\tau)| > m$, by (5) of Lemma 3.8.32, it is sufficient to show that $\mathit{Cost}(T_3) = O(\mathit{Cost}(S, \mu))$.

We first bound $\mathit{DropCost}(T_3)$ as follows. By (1) of Lemma 3.8.31, (1) of Lemma 3.8.32, and Fact 3.8.19, any pairs of schedules from S_4, S_5, T_2, and T_3 are prefix-matched. By (2) of Lemma 3.8.31, $\mathit{Set}(\mathit{suf}(S_4)) \subseteq \mathit{SecHvyJobs}(\tau)$. By (3) of Lemma 3.8.31, $(\mathit{Set}(\mathit{suf}(S_4)), S_4)$ satisfies the Greedy-Packing property. By (4) of Lemma 3.8.32, $(\mathit{SecHvyJobs}(\tau), T_2)$ satisfies the Greedy-Packing property. By (1) of Fact 3.8.7, $|\mathit{suf}(S_5)| \leq |\mathit{suf}(T_3)|$, and the total drop cost of the jobs in S_5 is at most that of the jobs in T_3. Since S_5 and T_3 are prefix-matched, $\mathit{DropCost}(T_3) \leq \mathit{DropCost}(S_5)$. By (4) of Lemma 3.8.31, $\mathit{DropCost}(T_3) = O(\mathit{Cost}(S, \mu))$.

We then bound $\mathit{IntReconfigCost}(T_3)$. By (4) of Lemma 3.8.32 and Lemma 3.8.22, $\mathit{IntReconfigCost}(T_3) = \Delta \cdot \Theta(|\mathit{hasSuf}(T_3)| + |\mathit{Colors}(\mathit{suf}(T_3))|)$. Thus, it is sufficient to bound $\mathit{hasSuf}(T_3)$ and $|\mathit{Colors}(\mathit{suf}(T_3))|$, respectively. By (2) of Fact 3.8.7, $|\mathit{hasSuf}(T_3)| \cdot \Delta \leq \mathit{DropCost}(S_5) + |\mathit{hasSuf}(S_5)| \cdot \Delta$. Since $|\mathit{hasSuf}(S_5)| \cdot \Delta \leq \mathit{IntReconfigCost}(S_5)$, $|\mathit{hasSuf}(T_3)| \cdot \Delta \leq \mathit{IntCost}(S_5)$. By (4) of Lemma 3.8.31, $|\mathit{hasSuf}(T_3)| \cdot \Delta = O(\mathit{Cost}(S, \mu))$.

By (3) of Lemma 3.8.32, $|\mathit{Colors}(\mathit{suf}(T_3))| \leq |\mathit{SecHvyColors}(\tau)|$. Since $|\mathit{HvyColors}(\tau)| > m$, $|\mathit{PrmyHvyColors}(\tau)| = m$ and $|\mathit{SecHvyColors}(\tau)| = |\mathit{HvyColors}(\tau)| - m$. By Lemma 3.8.21, $\mathit{Cost}(S, \mu) = \Omega(|\mathit{SecHvyColors}(\tau)|)$. Hence, $|\mathit{Colors}(\mathit{suf}(T_3))| \cdot \Delta = O(\mathit{Cost}(S, \mu))$. □

Lemma 3.8.34. *Consider any frame τ, any schedule S for τ, and any coloring μ. Let $T = \mathit{trunc}(\mathit{ReshapeFrame}(\tau, \nu))$, where $\nu = \mathit{extractedColoring}(\mu)$. If $|\mathit{HvyColors}(\tau)| > m$, then $\mathit{ExtReconfigCost}(T, \mu) = O(\mathit{Cost}(S, \mu))$.*

Proof. By definition, $\mathit{ExtReconfigCost}(T, \mu) = |\mathit{Mismatch}(()T, \mu)|$. By the definition of *extractedColoring*, $\mathit{Mismatch}(T, \mu) \subseteq \mathit{Mismatch}(T, \nu)$. Let $T_1 = \mathit{prefixHvy}(\tau, \mathit{PrmyHvyJobs}(\tau), \nu)$, $T_2 = \mathit{pack}(T_1, \mathit{SecHvyJobs}(\tau))$, and $T_3 = \mathit{dropSuf}(T_2)$. By (5) of Lemma 3.8.32, $T_3 = T$. By (2) of Lemma 3.8.32, T_1 and T_3 are prefix-identical. Hence, it is sufficient to show that $\mathit{Mismatch}(T_1, \nu) = O(\mathit{Cost}(S, \mu))$, which follows from Lemma 3.8.4. Hence, the lemma follows. □

Lemma 3.8.13 follows from Lemmas 3.8.33 and 3.8.34.

3.8.8 Few Heavy Colors

The purpose of this section is to provide the proof of Lemma 3.8.14 (stated in Section 3.8.6). Before that, we give the following lemma.

Lemma 3.8.35. *Consider any frame τ and any coloring μ. Let schedule $S = \mathit{prefixHvy}(\tau, \mathit{PrmyHvyColors}(\tau), \mu)$ and $T = \mathit{trunc}(\mathit{prefixLight}(S, \mu))$. If $|\mathit{HvyColors}(\tau)| \leq m$, then schedule T equals $\mathit{trunc}(\mathit{ReshapeFrame}(\tau, \mu))$.*

Proof. Let schedule $S' = \mathit{pack}(S, \mathit{SecHvyJobs}(\tau))$, and $S'' = \mathit{dropSuf}(S')$. Since $|\mathit{HvyColors}(\tau)| \leq m$, $\mathit{SecHvyJobs}(\tau) = \emptyset$. By (2) of Fact 3.8.3, S is suffix-free. By (1) of Fact 3.8.5, $S' = S$. By (3) of Fact 3.8.6, $S'' = S'$. The lemma then follows from the definition of *ReshapeFrame*. □

Proof of Lemma 3.8.14. Let $T_1 = \mathit{prefixHvy}(\tau, \mathit{PrmyHvyColors}(\tau), \nu)$, and $T_2 = \mathit{prefixLight}(T_1, \nu)$. By Lemma 3.8.35, it is sufficient to show that

$$\mathit{Cost}(T_2, \mu) = O(\mathit{Cost}(S, \mu)),$$

which we establish as follows.

We first consider $\mathit{ReconfigCost}(T_2, \mu)$. By (2) of Fact 3.8.3 and (1) of Fact 3.8.8, T_2 is suffix-free. Hence, $\mathit{IntReconfigCost}(T_2) = 0$. By definition, $\mathit{ExtReconfigCost}(T_2, \mu) = |\mathit{Mismatch}(T_2, \mu)|$. By the definition of *extractedColoring*, $\mathit{Mismatch}(T_2, \mu) \subseteq \mathit{Mismatch}(T_2, \nu)$.

We then consider $\mathit{DropCost}(T_2)$. By the definition of *prefixHvy*, all jobs in $\mathit{PrmyHvyColors}(\tau)$ are executed in T_1. By the definition of function *prefixLight*, T_2 does not drop any job executed in T_1. Hence, all jobs in $\mathit{PrmyHvyColors}(\tau)$ are executed in T_2. Since $\mathit{HvyColors}(\tau) \leq m$, by definition, $\mathit{PrmyHvyColors}(\tau) = \mathit{HvyColors}(\tau)$. Thus, for any color ℓ such that $\mathit{DropSet}(T_2, \ell) \neq \emptyset$, ℓ is a light color. By (4) of Fact 3.8.8, $\mathit{DropCost}(T_2) \leq \mathit{DropCost}(S) + \Delta \cdot \mathit{Mismatch}(T_2, \nu)$.

By (3) of Fact 3.8.8, we have $\mathit{Mismatch}(T_2, \nu) = \mathit{Mismatch}(T_1, \nu)$. By Lemma 3.8.4, $\mathit{Mismatch}(T_1, \nu) = O(\mathit{Cost}(S, \mu))$. Hence, $\mathit{ReconfigCost}(T_2, \mu)$ and $\mathit{DropCost}(T_2)$ are $O(\mathit{Cost}(S, \mu))$, respectively. □

3.9 Batched Arrivals

In this section we solve $[\Delta \mid d_\ell \mid D \mid D]$, which is characterized by a fixed reconfiguration cost Δ, per-color drop costs d_ℓ, a fixed delay bound D, and batched arrivals (jobs arrive at integral multiples of D).

As mentioned in Section 3.1, $[\Delta \mid d_\ell \mid D \mid D]$ is a building block to solve our main problem $[\Delta \mid d_\ell \mid D \mid 1]$. To solve $[\Delta \mid d_\ell \mid D \mid D]$, we use a reduction to rate-limited $[\Delta \mid d_\ell \mid D \mid D]$, which is solved in Section 3.7. Sections 3.9.1 and 3.9.2 give the reduction algorithm and analysis, respectively.

3.9.1 Algorithm *Recolor*

For any request r for $[\Delta \mid d_\ell \mid D \mid D]$, we define $\mathit{recolored}(r)$ as a request obtained as follows. For any color ℓ, we rank color ℓ jobs in r in an arbitrary order. For any color ℓ and color ℓ job x in r, we construct a job y that is the same as x except that the color of y is given by the pair (ℓ, j), where $j = \left\lfloor \frac{\mathit{rank}(x)}{D} \right\rfloor$, and $\mathit{rank}(x)$ is the rank of x in r. The request $\mathit{recolored}(r)$ is the union of all such y's that are constructed over all colors ℓ.

Consider any request sequence σ for $[\Delta \mid d_\ell \mid D \mid D]$. Let σ_i be request i of σ, where $0 \leq i < |\sigma|$. We obtain a request sequence $\mathit{recoloredReqSeq}(\sigma)$ by concatenating $\mathit{recolored}(\sigma_i)$'s in increasing order of i. It is not hard to see that $\mathit{recoloredReqSeq}(\sigma)$ is a request sequence for rate-limited $[\Delta \mid d_\ell \mid D \mid D]$.

Given any instance (σ, m) of $[\Delta \mid d_\ell \mid D \mid D]$, algorithm *Recolor* proceeds as follows. First, we use algorithm *Split* on (σ', m) to obtain a schedule S' for σ', where $\sigma' = \mathit{recoloredReqSeq}(\sigma)$. Second, from S' we construct a schedule S by replacing color (ℓ, j) with color ℓ in S', for any color ℓ and any nonnegative integer j. It is not hard to see that S is a schedule for σ.

3.9.2 Analysis of *Recolor*

In this section, we show that algorithm *Recolor* is resource competitive for $[\Delta \mid d_\ell \mid D \mid D]$. Before that, we first establish some preliminary results.

Consider any request sequence σ for $[\Delta \mid d_\ell \mid D \mid D]$ and any schedule S for σ. Given an initial coloring μ, the coloring of the resources at the beginning of block i is determined by S. As mentioned in Section 2.2, if the initial coloring is not specified, we assume the default coloring in which resources are colored black. Consider any block i. For any color ℓ, we define $\mathit{Mono}(S, i, \ell)$ as the set of resources k such that (1) the color of resource k at the beginning of block i is ℓ, and (2) all jobs executed on resource k in block i, if any, are color ℓ jobs. We define $\mathit{Mono}(S, i)$ as $\cup_\ell \mathit{Mono}(S, i, \ell)$. We define $\mathit{Multi}(S, i)$ as the set of resources not in $\mathit{Mono}(S, i)$. We define $\mathit{Full}(S, i)$ (resp., $\mathit{Empty}(S, i)$) as the set of resources k such that each slot in resource k in block i is occupied (resp. free) in S.

Lemma 3.9.1. *For any request sequence σ for $[\Delta \mid d_\ell \mid D \mid D]$, if there exists an m-resource schedule S for σ with cost C, then there exists an m-resource schedule for recoloredReqSeq(σ) with cost $O(C)$.*

Proof. We construct an m-resource schedule by initializing an m-resource schedule T as schedule S and then modifying T in the following two phases. In the first phase, we rearrange the job executions in T in increasing order of block indices. For any block i, we rearrange the job executions in block i in the following two stages. In the first stage, we rearrange the job executions in a way that jobs of the same color on the same resource in block i are executed contiguously. In the second stage, we rearrange the job executions in an arbitrary order of colors. For any color ℓ, we proceed iteratively. Each iteration proceeds as follows. Let $X_\ell = \mathit{Mono}(T, i, \ell) \setminus \mathit{Full}(T, i)$, and Y_ℓ be the set of resources k in $\mathit{Multi}(T, i)$ that executes at least one color ℓ job in block i in T. If $X_\ell = \emptyset$, we terminate the processing of color ℓ jobs. Otherwise, we proceed in the following steps. In the first step, we pick the resource k in X_ℓ with the smallest index. In the second step, if $Y_\ell \neq \emptyset$, we pick any resource p in Y_ℓ; otherwise if $|X_\ell| > 1$, we pick any resource p in X_ℓ such that $p \neq k$; otherwise, we terminate the processing of color ℓ. In the third step, let x be any color ℓ job executed on resource p. We move x to execute in any free slot in resource k.

In the second phase, we recolor the jobs executed in T in increasing order of block indices. For any block i, we recolor the jobs executed in block i in an arbitrary order of the colors. For any color ℓ, we proceed in the following three stages. In the first stage, we label the color ℓ jobs executed on the resources in $\mathit{Mono}(T, i, \ell)$ from 0 to $|\mathit{Mono}(T, i, \ell)| - 1$. In the following we describe a way to label the resources in $\mathit{Mono}(T, i, \ell)$; any job x executed on a resource k in $\mathit{Mono}(T, i, \ell)$ is assigned the label that is assigned to resource k. If $i = 0$, then we label the resources in $\mathit{Mono}(T, i, \ell)$ from 0 to $|\mathit{Mono}(T, i, \ell)| - 1$ arbitrarily. Otherwise, for any resource k in $\mathit{Mono}(T, i, \ell) \cap \mathit{Mono}(T, i-1, \ell)$ such that the label assigned to resource k in block $i - 1$ is in the range 0 to $|\mathit{Mono}(T, i, \ell)| - 1$, we let resource k inherit its label in block $i - 1$; we assign the remaining labels in the range 0 to $|Mono(T, i, \ell)| - 1$ to the other resources in $Mono(T, i, \ell)$ arbitrarily. In the second stage, we label the color ℓ jobs executed on the resources in $Multi(T, i)$ as follows. We obtain a sequence α_ℓ of color ℓ jobs by concatenating the sequence of color ℓ jobs on each resource in $Multi(T, i)$, in an arbitrary order of the resources. For any nonnegative integer j such that $0 \leq j < \left\lceil \frac{|\alpha_\ell|}{D} \right\rceil$, let $\alpha_{\ell,j}$ be the sequence of jobs that consists of jobs $j \cdot D$ through $\min((j+1) \cdot D - 1, |\alpha_\ell|)$ of α_ℓ. We assign the label $|Mono(T, i, \ell)| + j$ to each job in $\alpha_{\ell,j}$. In the third stage, we recolor the jobs executed in block i based on the labels assigned in the first two stages. For any color ℓ and any color ℓ job x, we recolor x as color (ℓ, j), where j is the label we assign to x in block i.

Let T_1 (resp., T_2) be the schedule that we obtain at the end of the first (resp., second) phase. We first show that T_2 is a schedule for the request sequence *recoloredReqSeq*(σ). Consider any block i. It is not hard to verify the following claim: For any color ℓ, $|Mono(T_1, i, \ell) \setminus Full(T_1, i)| \leq 1$, and if $|Mono(T_1, i, \ell) \setminus Full(T_1, i)| = 1$, no color ℓ jobs are executed on the resources in $Multi(T_1, i)$ in block i in T_1. From the above claim, and the way that we label and recolor jobs, the set of jobs executed in T_2 in block i is a subset of *recolored*(σ_i), where σ_i is request i of σ. Summing up over all i, T_2 is a schedule for *recoloredReqSeq*(σ).

We then bound $DropCost(T_2)$. Since we construct T_2 by rearranging and recoloring jobs, and the way that we recolor a job does not change the drop cost of the job, $DropCost(T_2) = DropCost(S)$.

Finally, we bound $\mathit{ReconfigCost}(T_2)$. It is not hard to see that the first phase does not increase the reconfiguration cost, that is, $\mathit{ReconfigCost}(T_1) \leq \mathit{ReconfigCost}(S)$. It remains to show that

$$\mathit{ReconfigCost}(T_2) = O(\mathit{ReconfigCost}(T_1)).$$

For the purpose of analysis, we assign credit as follows. Consider any block i. For each resource k in $\mathit{Multi}(T_1, i)$, we assign Δ units of "multichromatic" credit to resource k, and for each color ℓ such that at least one color ℓ job is executed on resource k in block i in T_1, we assign Δ units of "split" credit to color ℓ. It is not hard to see that jobs of the same color on the same resource in $\mathit{Multi}(T_1, i)$ in T_1 are recolored to at most two different colors in T_2. Hence, the total credit assigned in block i is at most twice the reconfiguration cost incurred by T_1 in block i.

By the method used to recolor jobs, in block i, T_2 incurs a reconfiguration in the following three cases. The first case, T_1 incurs a reconfiguration. The second case occurs when $i > 0$. In this case, for each color ℓ and each resource k in $\mathit{Mono}(T_1, i-1, \ell) \cap \mathit{Mono}(T_1, i, \ell)$ such that the labels assigned to resource k in blocks i and $i-1$ are different, T_2 incurs an extra reconfiguration on resource k in the first round of block i. In the third case, for each color ℓ and each resource k in $\mathit{Multi}(T_1, i)$ such that color ℓ jobs executed on resource k in block i in T_1 are recolored to two different colors in T_2, T_2 incurs an extra reconfiguration on resource k.

By the method used to label and recolor jobs, the number of extra reconfigurations in the second case equals

$$\sum_{\ell}(|Mono(T_1, i-1, \ell)| - |Mono(T_1, i, \ell)|).$$

It is not hard to see that

$$|Mono(T_1, i-1, \ell)| - |Mono(T_1, i, \ell)| \leq |Mono(T_1, i-1, \ell) \cap Multi(T_1, i)|.$$

Hence, in the second case, the number of extra reconfigurations is at most $|Multi(T_1, i)|$. By the method used to assign credit, the cost incurred by the extra reconfigurations in the second case is at most the multichromatic credit. By the method used to assign credit, the cost incurred by extra reconfigurations in the third case is at most the total split credit assigned in block i. Summing up over all nonnegative integers i, $ReconfigCost(T_2) = O(ReconfigCost(T_1))$. $\square$

Lemma 3.9.2. *Consider any instance (σ, m) for $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$. Let $\sigma' = recoloredReqSeq(\sigma)$. Let S' be the schedule produced by Split on (σ', m). Let S be the schedule produced by Recolor on (σ, m). Then S' uses the same number of resources as S and $Cost(S) \leq Cost(S')$.*

Proof. By the definition of algorithm *Recolor*, the schedule S is obtained from S' by replacing color (ℓ, j), for any nonnegative integer j, with color ℓ. Hence, S' uses the same number of resources as S, and $ReconfigCost(S) \leq ReconfigCost(S')$. By the definition of $recoloredReqSeq(\sigma)$, the number of color ℓ jobs appearing in σ equals the total number of color (ℓ, j) jobs appearing in σ', over all nonnegative integers j. Hence, $DropCost(S) = DropCost(S')$. $\square$

Theorem 3.5. *Algorithm Recolor is resource competitive for* $[\Delta \mid d_\ell \mid D \mid D]$.

Proof. Consider any instance (σ, m) of $[\Delta \mid d_\ell \mid D \mid D]$. Let T be the schedule produced by an arbitrary feasible offline algorithm on (σ, m). By the definition of a feasible algorithm, T uses m resources. Let $C = \mathit{Cost}(T)$ and $\sigma' = \mathit{recoloredReqSeq}(\sigma)$. By Lemma 3.9.1, there exists an m-resource schedule T' for σ' with cost $O(C)$. Let S' be the schedule produced by algorithm *Split* for (σ', m). By Theorem 3.4, S' uses $O(m)$ resources and incurs cost $O(C)$. Let S be the schedule produced by algorithm *Recolor* on (σ, m). The theorem then follows from Lemma 3.9.2. $\square$

3.10 Main Problem

In this section, we solve our main problem, $[\Delta \mid d_\ell \mid D \mid 1]$, which is characterized by a fixed configuration cost Δ, per-color drop costs d_ℓ, a fixed drop cost D, and nonbatched arrivals (jobs can arrive in any round). As indicated earlier, our solution to this problem uses a reduction to $[\Delta \mid d_\ell \mid D \mid D]$, which is solved in Section 3.9.

3.10.1 Algorithm *Batch*

In this section, we define algorithm *Batch*, which solves $[\Delta \mid d_\ell \mid D \mid 1]$. Before presenting the algorithm, let us first give some definitions. Given an arbitrary request sequence σ for $[\Delta \mid d_\ell \mid D \mid 1]$, we define $\mathit{batchedReqSeq}(\sigma)$ as a request sequence obtained by moving the arrival of any job x in σ that arrives in half-block i to the beginning of of half-block $i + 1$, and changing the delay bound of x to $\frac{D}{2}$. Thus, *batchedReqSeq*(σ) can be viewed as a request sequence for $[\Delta \mid d_\ell \mid \frac{D}{2} \mid \frac{D}{2}]$.

We define algorithm *Batch* as follows. First, given any instance (σ, m) of $[\Delta \mid d_\ell \mid D \mid 1]$, we construct an instance $(\sigma', 3m)$ of $[\Delta \mid d_\ell \mid \frac{D}{2} \mid \frac{D}{2}]$, where $\sigma' = \mathit{batchedReqSeq}(\sigma)$. Second, we run algorithm *Recolor* (defined in Section 3.9.1) on $(\sigma', 3m)$ to obtain a schedule S' for σ'. Finally, we obtain S for σ from S', where S is the same as S' except that the request sequence associated with S is σ. Note that algorithm *Batch* is an online algorithm.

3.10.2 Analysis of *Batch*

In this section, we show that algorithm *Batch* is resource competitive for $[\Delta \mid d_\ell \mid D \mid 1]$. Before doing so, we give some definitions and preliminary results.

Consider any request sequence σ for $[\Delta \mid d_\ell \mid D \mid 1]$, and any schedule T for σ. For any color ℓ and any color ℓ job x that arrives in half-block i in σ, we say that the execution of x in T is *σ-early* (resp., *σ-punctual*, *σ-late*) if x is executed in half-block i (resp., half-block $i+1$, half-block $i+2$) in T.

Lemma 3.10.1. *For any request sequence σ for $[\Delta \mid d_\ell \mid D \mid 1]$, and any schedule T for σ, if all job executions in T are σ-punctual, then T is also a schedule for batchedReqSeq(σ).*

Proof. The lemma follows immediately from the definitions of a σ-punctual execution and *batchedReqSeq*(σ). $\square$

Lemma 3.10.2. *For any request sequence σ for $[\Delta \mid d_\ell \mid D \mid 1]$, if there exists an m-resource schedule T for σ with cost C, then there exists a $3m$-resource schedule T' for σ such that all job executions in T' are σ-punctual, and $Cost(T') = O(C)$.*

Proof. We construct a $3m$-resource schedule T' as follows. We use the first (resp., second, third) m resources of T' to schedule only jobs whose executions are σ-early (resp., σ-punctual, σ-late) in T, where each σ-early execution made by T is postponed by $\frac{D}{2}$ rounds (resp., made in the same round, $\frac{D}{2}$ rounds earlier).

From the way we construct T', it is not hard to see that the set of jobs executed in T' is the same as that executed in T, and the reconfiguration cost incurred by T' in the first (resp., second, third) m resources is at most $ReconfigCost(T)$. Hence, $Cost(T') = O(Cost(T))$. It is straightforward to see that each job execution in T becomes a σ-punctual execution in T'. Hence, the lemma follows. □

Theorem 3.6. *Algorithm Batch is resource competitive for $[\Delta \mid d_\ell \mid D \mid 1]$.*

Proof. Consider an arbitrary instance (σ, m) of $[\Delta \mid d_\ell \mid D \mid 1]$. Let T be a schedule produced by any feasible offline algorithm on (σ, m). By the definition of a feasible algorithm, T uses m resources. Let $C = Cost(T)$, and $\sigma' = batchedReqSeq(\sigma)$. By Lemmas 3.10.1 and 3.10.2, there exists a $3m$-resource schedule T' for σ' with cost $O(C)$.

Let S (resp., S') be the schedule produced by *Batch* (resp., *Recolor*) on (σ, m) (resp., $(\sigma', 3m)$). Since σ' can be viewed as a request sequence for $[\Delta \mid d_\ell \mid \frac{D}{2} \mid \frac{D}{2}]$, and by Theorem 3.5, algorithm *Recolor* is resource competitive for $[\Delta \mid d_\ell \mid \frac{D}{2} \mid \frac{D}{2}]$, the schedule S' uses $O(m)$ resources and incurs cost $O(C)$. By the definition of algorithm *Batch*, the schedule S is the same as S' except that the associated request sequence is σ. Hence, the lemma follows. $\square$

Chapter 4

Reconfigurable Resource Scheduling with Variable Delay Bounds

4.1 Introduction

In this chapter, we present our solution to the problem of reconfigurable resource scheduling with variable delay bounds, that is, $[\Delta \mid 1 \mid D_\ell \mid 1]$. We give a resource competitive algorithm for this problem, where the competitive ratio that we obtain does not depend on the various problem parameters, that is, Δ and the D_ℓ's.

To appreciate some of the difficulties associated with variable delay bounds, consider a scenario in which we are scheduling two categories of jobs on a single resource: "background" jobs and "short-term" jobs. Background jobs have deadlines far in the future, and short-term jobs have smaller delay bounds and arrive intermittently. We need to decide whether to use idle cycles to execute background jobs. If we allow background jobs to use idle cycles whenever available, we may incur a large number of reconfigurations, or drop a lot of short-term jobs; later on, we may regret incurring these costs if we encounter a lengthy interval during which no short-term jobs arrive, and during which all of the background jobs could have been executed using a single reconfiguration. On the other hand, if we do not allow background jobs to use small chunks of idle cycles, and instead wait for a long idle interval, then later on, we may regret doing so if we never encounter a long idle interval. In summary, these two basic approaches lead to either *thrashing* (i.e., excessively high reconfiguration cost) or *underutilization* (i.e., excessively high drop cost).

A natural way to try to overcome these difficulties is to consider algorithms based on the Least Recently Used (LRU) principle. To pursue this approach, we need to define an appropriate notion of an LRU timestamp in the current setting. We have investigated various natural alternatives. (See Section 4.3.3 for an example.) For all of these alternatives, we encounter the following basic difficulty, even with resource augmentation: If we configure the categories with the most recent LRU timestamps without considering whether these categories have jobs to execute, then we are vulnerable to underutilization; if we configure the categories with the most recent LRU timestamps and with jobs to execute, then we are vulnerable to thrashing.

Another natural approach is to consider algorithms based on the Earliest Deadline First (EDF) principle. As with LRU, there are different ways that we can formulate a specific algorithm based on the EDF principle. (See Section 4.3.2 for an example.) However, even with resource augmentation, all EDF variants seem to suffer from thrashing, and therefore fail to yield a resource competitive solution. Furthermore, it is not hard to argue that similar scheduling principles, such as Least Slack First, also suffer from thrashing.

Though EDF alone or LRU alone seems insufficient to solve our problem, each maintains a dynamic ordering that addresses a key aspect of the request sequence. EDF addresses the urgency aspect and tends to reduce the drop cost. LRU addresses the recency aspect and tends to reduce the reconfiguration cost. Moreover, each dynamic ordering is efficiently maintainable. It is natural to ask whether we can efficiently combine these two orderings, and thereby address both key aspects of the request sequence. In this dissertation, we answer this question in the affirmative. We propose a natural and efficient combination of EDF and LRU. The main idea is to keep two sets of categories configured: one set selected by the EDF principle, and the other selected by the LRU principle. (See Section 4.3.4 for the formal definition of this combination.) We prove that this combination yields a resource competitive algorithm for reconfigurable resource scheduling with variable delay bounds. The combining mechanism that we use to combine EDF and LRU is general in nature, and can be used to combine multiple scheduling principles, each of which maintains a dynamic ordering of the jobs. The present work suggests that, for problems which cannot be solved by a single dynamic ordering, it is worthwhile to explore algorithms based on a combination of dynamic orderings.

We use a layered approach to solve reconfigurable resource scheduling with variable delay bounds. First, we use a batching subroutine to reduce $[\Delta \mid 1 \mid D_\ell \mid 1]$ to the special case in which jobs of a given category arrive at integral multiples of the category-specific delay bound; we refer to this problem as $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$. Second, we reduce $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$ to a rate-limited problem in which, for each color ℓ, at most D_ℓ jobs of color ℓ arrive at each integral multiple of D_ℓ, denoted rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$. Third, we solve rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$ using the aforementioned combination of EDF and LRU. The first two layers are analogous to the first two layers in our solution to reconfigurable resource scheduling with variable drop costs, but are more involved due to the variable delay bounds.

In this chapter, we make use of the following definitions. Consider any delay bound p and any nonnegative integer i. We define the block (resp., half-block) of delay bound p with index i, denoted $block(p, i)$ (resp., $halfBlock(p, i)$), as the p (resp., $\frac{p}{2}$) rounds starting from round $i \cdot p$ (resp., $i \cdot \frac{p}{2}$).

The remainder of this chapter is organized as follows. Section 4.2 discusses related work. Section 4.3 presents our solution to rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$. Section 4.4 presents our solution to $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$. Section 4.5 presents our solution to $[\Delta \mid 1 \mid D_\ell \mid 1]$.

4.2 Related Work

The EDF scheduling algorithm is shown to be an optimal preemptive uniprocessor scheduling algorithm for certain schedulability problems that do not involve reconfiguration overhead [11, 18]. In this proposal, we discuss the issues associated with applying EDF in the context of reconfigurable resource scheduling with variable delay bounds, and propose a combination of EDF and LRU to address these issues.

Sleator and Tarjan [28] shows that LRU is constant competitive when given a constant factor advantage in the cache size. O'Neil et al. [21] consider a variation of LRU called *LRU-k*, which keeps track of the times of the last k references to each page. Megiddo et al. [20] consider a self-tuning cache replacement policy called Adaptive Replacement Cache, which combines recency and frequency aspects of the request sequence by maintaining two lists: one list captures the recency aspect, and the other captures the frequency aspect. Our combination of EDF and LRU integrates recency and urgency aspects by keeping two sets of categories configured: one set captures the recency aspect and the other captures the urgency aspect.

4.3 Rate-Limited Batched Arrivals

In this section, we solve rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, where each D_ℓ is a power of 2. This problem is characterized by a fixed reconfiguration cost Δ, a unit drop cost, per-color delay bounds D_ℓ, batched arrivals (jobs of color ℓ arrive at integral multiples of D_ℓ), and rate-limited input (at most D_ℓ jobs of color ℓ arrive at each integral multiple of D_ℓ). As mentioned in Section 1, this problem is a key building block to solve our main problem, namely $[\Delta \mid 1 \mid D_\ell \mid 1]$.

In this section, we introduce three online algorithms: *EDF*, *ΔLRU*, and *ΔLRU-EDF*. In Section 4.3.1, we first present the common aspects of the three algorithms. For instance, due to the difference between the reconfiguration and drop costs, we do not configure a color until it has enough job arrivals.

Algorithm *EDF* is based on the EDF scheduling principle. The main idea is that, among the colors with enough job arrivals, we configure the colors with the earliest deadlines and with jobs to execute. Algorithm *EDF* addresses the urgency aspect of the request sequence. However, since it favors colors that have jobs to execute, *EDF* suffers from thrashing. See Section 4.3.2 for a detailed discussion of *EDF*.

Algorithm ΔLRU is based on the LRU scheduling principle. The main idea is that, among the colors with enough job arrivals, we configure the colors with the most recent timestamps. (For the formal definition of the timestamp of a color, see Section 4.3.3.) Algorithm ΔLRU addresses the recency aspect of the request sequence. However, since it does not consider whether colors have jobs to execute, ΔLRU suffers from underutilization. See Section 4.3.3 for a detailed discussion of ΔLRU.

Algorithm Δ*LRU-EDF* is a combination of *EDF* and ΔLRU. The *EDF* component ensures that the resources are well utilized. The ΔLRU component reduces thrashing by allowing colors with recent timestamps to remain configured. See Section 4.3.4 for the formal definition of Δ*LRU-EDF*, and Section 4.3.5 for a proof that Δ*LRU-EDF* is resource competitive.

4.3.1 Common Aspects

We find it convenient to view the set of resources as a cache, where resource k corresponds to cache location k. We view reconfiguring resource k with color ℓ as caching color ℓ at location k. We use a counting scheme to ensure that only colors with a sufficient number of job arrivals can be brought into the cache.

In the following, we formally present the common aspects of the three algorithms. Given an instance (σ, m) of rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, we allow an online algorithm to use n resources, where $n \geq m$. Each color is either *eligible* or *ineligible*. Only eligible colors can be brought into the cache. For each color, we maintain a counter and a deadline. Initially, the cache is empty, all colors are ineligible, and the counter and deadline associated with each color are zero. In each round j, the actions performed in the four phases are described as follows.

Arrival phase We receive a request. For any color ℓ, if j is an integral multiple of D_ℓ, we perform the following steps.

1. We increase the counter of ℓ by the number of color ℓ jobs received in this phase.
2. If the counter of ℓ is at least Δ, we set ℓ to eligible and reset the counter of ℓ.
3. We set the deadline of ℓ to $j + D_\ell$.

Reconfiguration phase We update the contents of the cache; the method used depends on the algorithm, see Section 4.3.2 through Section 4.3.4.

Execution phase For each color ℓ, we execute one pending job of color ℓ on each resource configured with color ℓ.

Drop phase For any color ℓ, if $j \bmod D_\ell$ is $D_\ell - 1$, we perform the following steps.

1. We drop all pending jobs of color ℓ.
2. If color ℓ is eligible and not in the cache, we set color ℓ to ineligible.

4.3.2 Algorithm *EDF*

We say that a color ℓ is *idle* if there are no pending jobs of color ℓ, and *nonidle* otherwise. We rank nonidle colors ahead of idle colors. The rank of idle colors is arbitrary. We rank nonidle colors in ascending order of the associated deadlines. Ties are broken in ascending order of the delay bounds. Further ties are broken according to a fixed order of colors. We update the cache as follows. If a nonidle eligible color ℓ in the top n positions of the ranking is not in the cache, we bring ℓ into the cache, evicting the color with the lowest rank if the cache is full.

Consider a color ℓ with a short delay bound that receives a small number of jobs every D_ℓ rounds. The priority of ℓ changes from high to low, and then low to high, from time to time, which may lead to thrashing. We refer the reader to Appendix A for an example establishing that *EDF* is not resource competitive.

4.3.3 Algorithm *ΔLRU*

For each color ℓ, we maintain a *timestamp* as follows. Initially, the timestamp of ℓ is zero. In the arrival phase of any round j, if the counter of ℓ

is reset, we set the timestamp of ℓ to j immediately after the counter is reset. In each reconfiguration phase, we cache the n eligible colors with the most recent timestamps, breaking ties as in EDF.

Due to the difference between the reconfiguration and drop costs, we require at least Δ jobs of color ℓ to arrive in order to update the timestamp of ℓ. Algorithm *ΔLRU* favors idle colors with recent timestamps over nonidle colors that do not have recent timestamps, which may result in low utilization. We refer the reader to Appendix B for an example establishing that *ΔLRU* is not resource competitive.

4.3.4 Algorithm *ΔLRU-EDF*

In this section, we formally define algorithm *ΔLRU-EDF*. We give *ΔLRU-EDF* a factor of 8 resource advantage over an optimal feasible offline algorithm, that is, we set $n = 8m$. We use the first half of the cache capacity to keep distinct colors and the remaining half to replicate the cache contents of the first half. We use the replication to give half of the resources a factor of 2 speedup. Below we describe how the first half of the cache is managed.

Let X be the $\frac{n}{4}$ eligible colors with the most recent timestamps, where ties are broken as in *ΔLRU*. We rank eligible colors not in X as in *EDF*. Let Y be the set of nonidle eligible colors in the top $\frac{n}{4}$ positions of the ranking. For any color ℓ that is in $X \cup Y$ but not in the cache, we bring ℓ into the cache, replacing an arbitrary color ℓ' that is in the cache but not in $X \cup Y$, if necessary. Since $|X \cup Y| \leq \frac{n}{2}$, such a color ℓ' is guaranteed to exist if the first half of the cache is full.

4.3.5 Analysis of *ΔLRU-EDF*

In this section, we show that *ΔLRU-EDF* is resource competitive for rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, where each D_ℓ is a power of 2. Before we give the details of our analysis, we offer a high level overview. The formal definitions mentioned in the overview are provided later in this section.

The analysis is organized as follows. First, Lemmas 4.3.1 and 4.3.2 establish some properties of eligible and ineligible jobs and colors. Second, Lemmas 4.3.3 through 4.3.6 argue that, on any instance such that each color appearing in the request sequence has at least Δ jobs, the cost incurred by *ΔLRU-EDF* is within a constant factor of that incurred by an optimal feasible offline algorithm. For convenience of analysis, we partition the drop costs incurred by *ΔLRU-EDF* into "eligible" and "ineligible" drop costs. Lemma 4.3.3 bounds the eligible drop cost of *ΔLRU-EDF*. Our proof of Lemma 4.3.3 uses the EDF properties of *ΔLRU-EDF*, and three intermediate algorithms: "parallel" *EDF*, denoted *Par-EDF*, "sequential" *EDF*, denoted *Seq-EDF*, and "double-speed" *Seq-EDF*, denoted *2X-Seq-EDF*.

To bound the other costs incurred by *ΔLRU-EDF*, for each color ℓ, we partition the sequence of rounds into subsequences, denoted "ℓ-epochs". Lemma 4.3.4 gives an upper bound on the ineligible drop cost incurred by *ΔLRU-EDF*, in terms of the total number of ℓ-epochs, over all colors ℓ. The proof of Lemma 4.3.4 is straightforward. For convenience of analysis, we label each eviction performed by *ΔLRU-EDF* as either an "LRU eviction" or an "EDF eviction". Lemma 4.3.5 bounds the total number of LRU evictions, and is invoked in the proof of Lemma 4.3.7. The proof of Lemma 4.3.5 uses the LRU properties of *ΔLRU-EDF*. For any problem instance such that each color appearing in the request sequence has at least Δ jobs, Lemma 4.3.6 lower bounds the total cost incurred by an optimal feasible offline algorithm, in terms of the total number of ℓ-epochs, over all colors ℓ; Lemma 4.3.7 upper bounds the reconfiguration cost incurred by *ΔLRU-EDF*, in terms of the cost incurred by an optimal feasible offline algorithm, and the total number of ℓ-epochs, over all colors ℓ. Our proofs of Lemmas 4.3.6 and 4.3.7 make use of amortized analysis; our proof of Lemma 4.3.6 relies on the LRU properties of *ΔLRU-EDF*.

Third, Theorem 4.1 establishes that *ΔLRU-EDF* is resource competitive by a reduction to a problem instance in which each color appearing in the request sequence has at least Δ jobs, and by using Lemmas 4.3.3 through 4.3.6.

Now we give the formal definitions used in our the analysis. Consider any instance (σ, m) of rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$. We say that a job x of color ℓ is *ineligible* (resp., *eligible*) if color ℓ is ineligible (resp., eligible) at the end of the arrival phase in which x arrives. We define the ineligible (resp., eligible) drop cost of *ΔLRU-EDF*, denoted *IneligibleDropCost*(*ΔLRU-EDF*$, \sigma, m$) (resp., *EligibleDropCost*(*ΔLRU-EDF*$, \sigma, m$)), as the drop cost incurred by *ΔLRU-EDF* on ineligible (resp., eligible) jobs in σ.

Lemma 4.3.1. *For any eligible (resp., ineligible) job x of color ℓ, color ℓ is* *eligible (resp., ineligible) from the end of the arrival phase in which x arrives until the deadline of job x is reached.*

Proof. The lemma follows from the definition of an eligible job, and the way we determine which colors are eligible. $\square$

Lemma 4.3.2. *All ineligible jobs are dropped by ΔLRU-EDF.*

Proof. From the definition of ΔLRU-EDF, it is not hard to see that only eligible colors can be cached by ΔLRU-EDF. The corollary then follows from Lemma 4.3.1 and the definition of an ineligible job. $\square$

For each color ℓ, we partition the sequence of rounds into ℓ-epochs as follows. We define ℓ-epoch 0 to start with round 0 and end with the first round in which ℓ becomes ineligible. For every $i \geq 1$, ℓ-epoch i starts when ℓ-epoch $i-1$ ends, and ends with the first round following ℓ-epoch $i-1$ in which ℓ becomes ineligible. For convenience, we use the term *epoch* to refer to an ℓ-epoch, for some ℓ. We use *numEpochs*(σ) to denote the total number of epochs associated with σ.

Lemma 4.3.3. *For any instance (σ, m) of rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$,*

$$\textit{EligibleDropCost}(\Delta\textit{LRU-EDF}, \sigma, m) \leq \textit{DropCost}(\textit{OFF}, \sigma, m).$$

Proof. See Section 4.3.6. $\square$

Lemma 4.3.4. *For any instance (σ, m) of rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$,*

$$\textit{IneligibleDropCost}(\Delta\textit{LRU-EDF}, \sigma, m) < \textit{numEpochs}(\sigma) \cdot \Delta.$$

Proof. Consider any color ℓ. Let h be any ℓ-epoch. Let C be the ineligible drop cost incurred by *ΔLRU-EDF* on color ℓ jobs in h. It is sufficient to show that C is less than Δ.

Let h' be the longest prefix of h throughout which ℓ is ineligible. Let C' be the drop cost incurred by *ΔLRU-EDF* on color ℓ jobs in h'. Since ℓ does not become eligible in h', the number of color ℓ jobs that arrive in h' is less than Δ. Hence, $C' < \Delta$. By the definition of an epoch, once ℓ becomes eligible in h, it remains eligible until h ends. By the definition of ineligible jobs and ineligible drop cost, $C = C'$. Therefore, $C < \Delta$. □

We find it useful to label each eviction by *ΔLRU-EDF* as either an "LRU eviction" or an "EDF eviction". We say that an LRU eviction occurs whenever a color is evicted in a given round and that color was kept by the LRU principle in the preceding round. All other evictions are EDF evictions. For any algorithm A, let $nEvictLRU(A, \sigma, m)$ be the number of LRU evictions performed by A on (σ, m).

Lemma 4.3.5. *For any instance (σ, m) of rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$ such that each D_ℓ is a power of 2, and each color has at least Δ job arrivals, $nEvictLRU(\Delta LRU\text{-}EDF, \sigma, m) \cdot \Delta$ is $O(Cost(OFF, \sigma, m))$.*

Proof. See Section 4.3.7. □

Lemma 4.3.6. *For any instance (σ, m) of rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$ such that each D_ℓ is a power of 2, and each color appearing in σ has at least Δ jobs, $Cost(OFF, \sigma, m) = \Omega(numEpochs(\sigma) \cdot \Delta)$.*

Proof. See Section 4.3.7. □

Lemma 4.3.7. *For any instance (σ, m) of rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$ such that each D_ℓ is a power of 2, and each color appearing in σ has at least Δ jobs,*

$$\mathit{ReconfigCost}(\mathit{\Delta LRU\text{-}EDF}, \sigma, m) \leq O(\mathit{Cost}(\mathit{OFF}, \sigma, m) + \mathit{numEpochs}(\sigma) \cdot \Delta).$$

Proof. See Section 4.3.8. □

Theorem 4.1. *Algorithm ΔLRU-EDF is resource competitive for rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, where each D_ℓ is a power of 2.*

Proof. Let (σ, m) be an arbitrary instance of rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$. We say that a color ℓ is *heavy* (resp., *light*) if there are at least (resp., less than) Δ jobs of color ℓ in σ. Any job of a heavy (resp., light) color is a heavy (resp., light) job. We break each request into two requests, one consisting of the light jobs and the other consisting of the heavy jobs. Let α (resp., β) denote the resulting sequence of requests involving heavy (resp., light) jobs.

Since there are less than Δ jobs of any light color, *OFF*, as an optimal feasible offline algorithm, drops all light jobs. Hence, $\mathit{Cost}(\mathit{OFF}, \sigma, m)$ equals $\mathit{Cost}(\mathit{OFF}, \alpha, m)$ plus the total number of light jobs. Since there are are less than Δ jobs of any light color, no light color ever becomes eligible. Thus, *ΔLRU-EDF* never caches a light color, and drops all light jobs. Hence, $\mathit{Cost}(\mathit{\Delta LRU\text{-}EDF}, \sigma, m)$ equals $\mathit{Cost}(\mathit{\Delta LRU\text{-}EDF}, \alpha, m)$ plus the total number of light jobs. From Lemmas 4.3.3, 4.3.4, 4.3.6, and 4.3.7, *Cost*($\mathit{\Delta LRU\text{-}EDF}, \alpha, m$) $= O(\mathit{Cost}(\mathit{OFF}, \alpha, m))$. Hence, the theorem follows. □

4.3.6 Eligible Drop Cost of *ΔLRU-EDF*

The purpose of this section is to provide the proof of Lemma 4.3.3. Before that, we first give some definitions and preliminary results.

We say that a schedule S is *double-speed* if the reconfiguration and execution phases are performed twice (resp., only once) in each round in schedule S. We say that an algorithm A is *double-speed* if for any input, algorithm A produces a double-speed schedule. We define a *mini-round* as an iteration of the reconfiguration and execution phases in a round. By definition, there are two mini-rounds in each round of a double-speed schedule. For a double-speed schedule, each slot (defined in Section 2.2) corresponds to a mini-round. For any mini-round, we define a *column* as the union of the slots that correspond to the mini-round. We say a column is *full* if all slots in the column are occupied, and *nonfull* otherwise.

Throughout this section, we find it useful to rank jobs as we rank colors in *EDF*. We define the following three algorithms: *Par-EDF*, *Seq-EDF*, and *2X-Seq-EDF*. Consider any instance (σ, m) of rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$. For each of these three algorithms, we allow m resources to be used. Algorithm *Par-EDF* is defined as follows. In each reconfiguration phase, we reconfigure the resources in such a way that we can execute m pending jobs with the best ranks in the immediately following execution phase. Algorithm *Seq-EDF* is defined as follows. In each reconfiguration phase, we configure m nonidle colors with the best ranks, where colors are ranked as in *EDF*. We use *2X-Seq-EDF* to denote double-speed *Seq-EDF*. Note that the three algorithms defined in this paragraph do not require a color to be eligible in order to be configured on the resources. We say that a request sequence σ is *Par-EDF-friendly* if *Par-EDF* does not incur any drops on (σ, m).

Lemma 4.3.8. *For any instance* (σ, m) *of rate-limited* $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, $\mathit{DropCost}(\mathit{Par\text{-}EDF}, \sigma, m) \leq \mathit{DropCost}(\mathit{OFF}, \sigma, m)$.

Proof. We view m resources as one super resource which can execute m jobs per round. The proof then follows from the optimality of the traditional EDF algorithm. □

Lemma 4.3.9. *For any instance* (σ, m) *of rate-limited* $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$ *such that* σ *is Par-EDF-friendly, algorithm 2X-Seq-EDF does not incur any drops on* (σ, m).

Proof. Let S denote the schedule produced by algorithm *Par-EDF* on (σ, m). We prove this lemma by constructing a schedule that executes the same set of jobs executed in S, and show that *2X-Seq-EDF* produces such a schedule.

We initialize a schedule T as a double-speed schedule that does not execute any jobs in σ, and then modify T by assigning the jobs in σ to execute in T as follows. We assign the jobs in increasing order of delay bounds. For a given delay bound p, we assign the jobs with delay bound p in increasing order of block (of delay bound p) indices. In any block of p, we assign the jobs with delay bound p according to a fixed order of colors as in *EDF*. Consider any delay bound p, any nonnegative integer i, and any color ℓ such that $D_\ell = p$. Let X_ℓ be set of color ℓ jobs that arrive in $block(p, i)$. In the remainder of this paragraph, we describe how we assign the jobs in X_ℓ, which we do in three steps. In the first step, we pick the first $|X_\ell|$ nonfull columns. (We will establish the existence of such columns in the next paragraph.) In the second step, in each of the columns picked in the first step, we pick an arbitrary free slot. In the third step, we assign the jobs in X_ℓ in the $|X_\ell|$ slots picked in the second step. Since all delay bounds are powers of 2, it is not hard to see that the schedule produced by *2X-Seq-EDF* is among the schedules that can be constructed using the above procedure.

It remains to show the following claim: There are at least $|X_\ell|$ nonfull columns before we arrange the jobs in X_ℓ. Let X_ℓ^* be the set of jobs that arrive in $block(p, i)$ and have higher ranks than the jobs in X_ℓ. From the way we arrange jobs, and the fact that all delay bounds are powers of 2, only the jobs in X_ℓ^* can be arranged in $block(p, i)$ before we arrange the jobs in X_ℓ. Since σ is *Par-EDF*-friendly, and all delay bounds are powers of 2, all jobs in X_ℓ^* are executed in S in $block(p, i)$. Hence $|X_\ell^*| \leq p \cdot m$. Since T is a double-speed schedule, and *Par-EDF* is a single-speed algorithm, the number of slots in $block(p, i)$ in T is twice that in S. Hence, before we arrange the jobs in X_ℓ in T, the following condition holds: In schedule T, at least $p \cdot m$ slots in $block(p, i)$ are free. Hence, in schedule T, at least p columns in $block(p, i)$ are nonfull.

By the definition of rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, $|X_\ell| \leq p$. Hence, the claim follows.

Lemma 4.3.10. *For any instance (σ, m) of rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$ and any subsequence α of σ, if 2X-Seq-EDF executes (resp., drops) j jobs on (α, m), then 2X-Seq-EDF executes (resp., drops) at least j jobs on (σ, m).*

Proof. Let β be the set of jobs that appear in σ and not in α. We number the jobs in β from zero in increasing order of arrival rounds, breaking ties arbitrarily. We define $\gamma_0 = \alpha$. For $0 \leq i < |\beta|$, we define β_i as job i in β and $\gamma_{i+1} = \gamma_i \cup \{\beta_i\}$. By definition, $\sigma = \gamma_{|\beta|}$.

For any integer i such that $0 \leq i < |\beta|$, we use S_i to denote the schedule produced by algorithm *2X-Seq-EDF* on (γ_i, m). In the following, we prove the lemma by showing that, for any integer i such that $0 \leq i < |\beta|$, $|X_i| \leq |X_{i+1}|$ and $|Y_i| \leq |Y_{i+1}|$, where X_i (resp., Y_i) is the set of jobs executed (resp., dropped) in S_i. We first consider the case where $\beta_i \notin X_{i+1}$. In this case, it is not hard to see that $X_{i+1} = X_i$ and $Y_i \subseteq Y_{i+1}$. We then consider the case where $\beta_i \in X_{i+1}$. Suppose β_i is executed in column j in S_{i+1}. Let $X_{i,j}$ be the set of jobs in X_i that are executed in column j in S_i, and $Z_{i,j}$ be set of slots used by $X_{i,j}$ in S_i. By the definition of *2X-Seq-EDF*, if $|Z_{i,j}| < m$, then $X_{i,j} \subseteq X_{i+1,j}$; if $|Z_{i,j}| = m$, then the job x with the lowest rank in $X_{i,j}$ is transferred to execute in the next column if the deadline of x has not been reached in the next column, and dropped otherwise. Similar transfers continue until either (1) the transferred job finds an empty slot in the next column, or (2) a job in the next column is dropped. Hence, either $X_i \subseteq X_{i+1}$ and $Y_i = Y_{i+1}$, or $|X_i| = |X_{i+1}|$ and $Y_i \subseteq Y_{i+1}$. This completes the proof. $\square$

Lemma 4.3.11. *For any instance (σ, m) of rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, DropCost(2X-Seq-EDF, σ, m) $\leq$ DropCost(Par-EDF, σ, m).*

Proof. If σ is *Par-EDF*-friendly, then the lemma follows immediately from Lemma 4.3.9. Otherwise, we break σ into two subsequences α and β, where α consists of the jobs executed by *Par-EDF* on (σ, m), and β consists of the remaining jobs, that is, the jobs dropped by *Par-EDF* on (σ, m). By Lemma 4.3.9, *2X-Seq-EDF* executes all jobs in (α, m). By Lemma 4.3.10, the number of jobs executed by *2X-Seq-EDF* on (σ, m) is at least the number of jobs executed by *2X-Seq-EDF* on (α, m). Hence the lemma follows. $\square$

Lemma 4.3.12. *For any instance (σ, m) of rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$,*

$$\textit{EligibleDropCost}(\textit{ΔLRU-EDF}, \sigma, m) \leq \textit{DropCost}(\textit{2X-Seq-EDF}, \sigma, m).$$

Proof. Let α be the subsequence of σ that consists of the eligible jobs in σ. By Lemma 4.3.10, it is sufficient to show that

$$\textit{EligibleDropCost}(\textit{ΔLRU-EDF}, \sigma, m) \leq \textit{DropCost}(\textit{2X-Seq-EDF}, \alpha, m),$$

which we argue as follows.

For convenience, we add a dummy round, denoted round -1, which only contains a dummy drop phase. For $-1 \leq i < |\sigma|$, let X_i (resp., Y_i) be the set of eligible pending jobs in *ΔLRU-EDF* (resp., *2X-Seq-EDF*) at the beginning of the drop phase in round i, and let X'_{i+1} (resp., X''_{i+1}) and Y'_{i+1} (resp., Y''_{i+1}) be the set of eligible pending jobs in *ΔLRU-EDF* (resp., *2X-Seq-EDF*) at the beginning and end of the arrival phase in round $i+1$, respectively. It is sufficient to show that for $-1 \leq i < |\sigma|$, $X_i \subseteq Y_i$, which we prove below by induction.

It is obvious that $X_{-1} = Y_{-1} = \emptyset$. Hence $X_{-1} \subseteq Y_{-1}$. We show in the following that, $X_i \subseteq Y_i$, for some $-1 \leq i < |\sigma| - 1$, implies $X_{i+1} \subseteq Y_{i+1}$. Since $X_i \subseteq Y_i$, and in both algorithms, jobs that reach their deadlines are dropped in drop phase $i+1$, $X'_{i+1} \subseteq Y'_{i+1}$. By this observation, and the fact that α only consists of the eligible jobs in σ, we have $X''_{i+1} \subseteq Y''_{i+1}$.

Let color ℓ be any color that is ever configured by *2X-Seq-EDF* in round $i+1$. Since *2X-Seq-EDF* is a double-speed schedule, at the end of the arrival phase in round $i+1$, color ℓ is among the $2m$ nonidle colors with the best ranks, and *2X-Seq-EDF* executes up to 2 jobs of color ℓ in round $i+1$. By Lemma 4.3.1, and the fact that α only consists of eligible jobs, in the reconfiguration phase of round $i+1$, color ℓ is eligible in *ΔLRU-EDF*. In *ΔLRU-EDF*, unless color ℓ is idle (which indicates all color ℓ jobs have been executed), color ℓ is also among the $2m$ nonidle eligible colors with the best ranks. Since $n = 8m$, i.e., $2m = \frac{n}{4}$, by the definition of *ΔLRU-EDF*, *ΔLRU-EDF* configures color ℓ in round $i+1$ and executes 2 jobs of color ℓ if there are at least 2, and all color ℓ jobs otherwise. By Lemma 4.3.2, *ΔLRU-EDF* only executes eligible jobs, that is, *ΔLRU-EDF* only executes jobs from X''_{i+1}. Since $X''_{i+1} \subseteq Y''_{i+1}$, we conclude that $X_{i+1} \subseteq Y_{i+1}$. □

Proof of Lemma 4.3.3. Immediate from Lemmas 4.3.8, 4.3.11, and 4.3.12. □

4.3.7 Lower Bound on the Cost of *OFF*

The purpose of this section is to provide the proof of Lemmas 4.3.5 and 4.3.6. First give some definitions and preliminary results.

Let (σ, m) be any instance of rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, where each D_ℓ is a power of 2. We find it convenient to partition the sequence of rounds into *super-epoch*s. Super-epoch 0 is the minimum sequence of rounds, starting with round 0, during which the counters of at least $2m$ colors are reset. For every positive integer i, super-epoch i is the minimum sequence of rounds following super-epoch $i-1$ during which the counters of at least $2m$ colors are reset. Note that the last super-epoch may be incomplete.

We define a color ℓ as an *i-active color* if the counter of ℓ is reset in super-epoch i, or in other words, the timestamp of ℓ is updated in super-epoch i. For any i-active color ℓ, an ℓ-epoch that overlaps with super-epoch i is referred to as an *i-active epoch*. We say that an epoch is *regular* if the epoch is complete and does not overlap with any incomplete super-epoch. Any epoch that is not regular is *special*.

For any color ℓ, we define a counter reset event of ℓ as the event that the counter of ℓ is reset. To simplify the presentation of our credit assignment rules (to be described later in this section), we attribute jobs to the counter reset events as follows. For any job x of color ℓ that arrives in round j, x is attributed to the counter reset event of ℓ that occurs in round k, where k is the smallest integer such that $k \geq j$ and the counter of ℓ is reset in round k. Note that not all the jobs are necessarily attributed to a counter reset event.

The following lemma follows from the way that we update counters, the definition of counter reset events, and the way that we attribute jobs to the counter reset events.

Lemma 4.3.13. *The number of jobs attributed to each counter reset event is at least* Δ.

We associate credit with the counter reset events as follows: (1) if color ℓ is i-active and there is a reconfiguration from or to color ℓ incurred by OFF in super-epoch i, we associate 4Δ units of credit with the first counter reset event of color ℓ in super-epoch i; (2) for each reconfiguration from or to a color ℓ incurred by OFF, we associate 4Δ units of credit with the next counter reset event of color ℓ; (3) for any color ℓ job x dropped by OFF, we associate 4 units of credit with the counter reset event to which x is attributed, if such an event exists.

The following lemma follows from the way that we assign credit.

Lemma 4.3.14. *The total credit associated with the counter reset events over all colors is* $O(Cost(OFF, \sigma, m))$.

Lemma 4.3.15. *For any i-active color ℓ, either ℓ is cached throughout super-epoch i by OFF, or there are at least 4Δ units of credit associated with the first counter reset events of ℓ in super-epoch i.*

Proof. Since color ℓ is i-active, the time stamp of ℓ is updated in super-epoch i, and hence super-epoch i is not empty. Let round j be the first round in super-epoch i. Let u be the first counter reset event in super-epoch i. We define integer k as follows. If there exists a counter reset event v prior to u, k is the index of the round in which v occurs. Otherwise, k is 0. In either case, $k \leq j$.

Let V be the sequence of rounds lying strictly between rounds k and j. We now prove the lemma as follows. If *OFF* evicts ℓ from the cache or loads ℓ into the cache in super-epoch i, by credit assignment rule (1), u is associated with 4Δ units of credit. If *OFF* keeps ℓ out of the cache throughout super-epoch i, we consider the following cases.

- In the first case, the interval V is not empty and algorithm *OFF* evicts ℓ out of the cache or loads ℓ into the cache in V. It is not hard to see that u is the first counter reset event following any reconfiguration in V. By credit assignment rule (2), u is associated with 4Δ units of credit.

- In the second case, the interval V is empty or algorithm *OFF* keeps ℓ out of the cache in V. Let k' be the round in which u occurs. In this case, *OFF* keeps ℓ out of the cache from the end of round k until the end of round k'. Since color ℓ jobs only arrive at an integral multiple of D_ℓ, all pending jobs of color ℓ are dropped in a round q such that $q \bmod D_\ell = D_\ell - 1$. By the definition of *ΔLRU-EDF*, the timestamp of color ℓ can only be updated at integral multiple of D_ℓ. From the way that jobs are attributed to counter reset events, all jobs attributed to u are dropped by *OFF*. By Lemma 4.3.13 and credit assignment rule (3), u is associated with at least 4Δ units of credit.

Hence, either ℓ is cached throughout super-epoch i, or u, the first counter reset event of ℓ in super-epoch i, is associated with at least 4Δ units of credit. □

Lemma 4.3.16. *For any color ℓ and any complete ℓ-epoch h, there exists a round j in h such that in the arrival phase of round j, the timestamp of ℓ is updated to j.*

Proof. Since h is complete, ℓ becomes eligible in h. Let s be the (arrival) phase in which ℓ becomes eligible. Let round j be the round that contains s. By the way that we update the counters and timestamps, the counter of ℓ is reset in s, and the timestamp of ℓ is updated in s to j. □

Lemma 4.3.17. *Consider any i-active color ℓ. Let j be the smallest integer such that round j is in super-epoch i and the timestamp of ℓ is updated in round j. Let round k be the last round in super-epoch i. If $j < k$, then color ℓ is kept in the cache by ΔLRU-EDF from the end of the reconfiguration phase of round j until the beginning of round k.*

Proof. Let V be the prefix of super-epoch i that includes all the rounds in super-epoch i except round k. Let X be the set of colors that update their timestamps in V. Since $j < k$, $\ell \in X$. By the definition of a super-epoch, $|X| < 2m$. Since a color always updates the timestamp to the index of the current round, ℓ is among the $2m$ colors with the most recent timestamps throughout the reconfiguration phase of round j until the beginning of round k. Hence the lemma follows. $\square$

Lemma 4.3.18. *For any nonnegative integer i, any i-active color is loaded into the cache by the LRU principle within ΔLRU-EDF at most once in super-epoch i.*

Proof. It is straightforward to see that any color ℓ can only be loaded into the cache by the LRU principle right after the timestamp of ℓ is updated. The lemma then follows by Lemma 4.3.17. $\square$

Lemma 4.3.19. *For any super-epoch i and any color ℓ, once super-epoch i contains a complete ℓ-epoch h, super-epoch i ends.*

Proof. We need to show that, as h ends, super-epoch i has ended. Let round k be the last round of super-epoch i. By Lemma 4.3.16, there exists a round j in h such that the timestamp of ℓ is updated to j. By Lemma 4.3.17, if $j < k$, ℓ is kept in the cache since the end of the reconfiguration phase of round j until the beginning of round k. Since an ℓ-epoch can only end when ℓ is ineligible, and only colors out of the cache can be ineligible, epoch h ends with round k. Hence, the lemma follows. $\square$

The next lemma follows immediately from Lemma 4.3.19.

Lemma 4.3.20. *For any color ℓ and any nonnegative integer i, there are at most two ℓ-epochs that overlap with super-epoch i.*

Lemma 4.3.21. *For each color ℓ, there are at most two special ℓ-epochs.*

Proof. By definition, a special epoch is either incomplete, or overlaps with an incomplete super-epoch. The lemma then follows from Lemma 4.3.20 and the fact that only the last ℓ-epoch and super-epoch can be incomplete. □

Lemma 4.3.22. *For any instance (σ, m) of rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, where each D_ℓ is a power of 2, and such that each color appearing in σ has at least Δ jobs, the cost incurred by OFF on any color is at least Δ.*

Proof. Consider any color ℓ. If *OFF* ever configures color ℓ, *OFF* incurs a cost of Δ. Otherwise, *OFF* drops all color ℓ jobs, incurring a cost of at least Δ since there are at least Δ color ℓ jobs in σ. In either case, *OFF* incurs at least a cost of Δ on color ℓ jobs. □

Lemma 4.3.23. *For any instance (σ, m) of rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, where each D_ℓ is a power of 2, and such that each color appearing in σ has at least Δ jobs, $Cost(OFF, \sigma, m)$ is at least $\frac{1}{2}\Delta$ times the total number of special epochs.*

Proof. The lemma follows immediately from Lemmas 4.3.21 and 4.3.22. □

Lemma 4.3.24. *The total credit associated with the counter reset events is at least $\frac{1}{4}\Delta$ times the total number of regular epochs.*

Proof. Let $X = \{i \mid \text{super-epoch } i \text{ is complete}\}$. Consider any $i \in X$. Let k_i be the number of i-active colors. Let k'_i be the number of i-active colors for which the first counter reset event in super-epoch i is associated with at least 4Δ units of credit. Let k''_i be the number of i-active colors that are cached throughout super-epoch i. By Lemma 4.3.15, $k_i \leq k'_i + k''_i$. By the definition of a super-epoch, $k_i \geq 2m$. Since $k''_i \leq m$, $k'_i \geq \frac{1}{2}k_i$, or in other words,

$$k_i \leq 2k'_i. \tag{4.3.1}$$

For any color ℓ, let $q_{i,\ell}$ denote the number of i-active ℓ-epochs and q_i be the number of i-active epochs.

$$\begin{aligned} \text{number of regular epochs} &\leq \sum_{i \in X} q_i \\ &= \sum_{i \in X} \sum_{\ell} q_{i,\ell} \\ &\leq 2 \sum_{i \in X} k_i \\ &\leq 4 \sum_{i \in X} k'_i. \end{aligned}$$

(The first inequality follows from the definitions of i-active epochs and regular epochs. The second equation is trivial. The third inequality follows from the definitions of i-active colors, i-active epochs and Lemma 4.3.20. The last inequality uses Equation (4.3.1).) By the definition of k'_i, the total credit is at least $4\Delta \sum_{i \in X} k'_i$. Hence the lemma follows. □

Lemma 4.3.6 follows from Lemmas 4.3.14, 4.3.23, and 4.3.24.

Proof of Lemma 4.3.5. For any positive integer i, let k_i be the number of i-active colors in super-epoch i. We prove the lemma in two stages. In the first stage, we show the following claim: In any super-epoch i, k_i is at least the number of LRU evictions performed by *ΔLRU-EDF*. By definition, any color that is not i-active does not update its timestamp in super-epoch i, and hence cannot result in any LRU evictions. The claim then follows from Lemma 4.3.18.

In the second stage, we show that $\sum_i k_i\Delta$ is $O(Cost(OFF, \sigma, m))$. Since only the last super-epoch can be incomplete, we consider the following two cases. In the first case, there is only one super-epoch, and this super-epoch is incomplete. In this case, Lemma 4.3.22 implies $Cost(OFF, \sigma, m) = \Omega(k_0\Delta)$. In the second case, there are at least two super-epochs. Let $Z = \{i \mid \text{super-epoch } i \text{ is complete}\}$. In this case, $\sum_i k_i = O(\sum_{i\in Z} k_i)$. By Lemma 4.3.14, it is sufficient to show that the total credit is $\Omega(\sum_{i\in Z} k_i\Delta)$. For any $i \in Z$, and any i-active color ℓ, Lemma 4.3.15 implies that there are at least $k_i - m = \Omega(k_i)$ i-active colors such that each of these colors ℓ has at least 4Δ units of credit associated with the first counter reset event of ℓ in super-epoch i. Hence, the total credit is $\Omega(\sum_{i\in Z} k_i\Delta)$. □

4.3.8 Reconfiguration Cost of *ΔLRU-EDF*

The purpose of this section is to provide the proof of Lemma 4.3.7. First we give some definitions and preliminary results.

In stating and proving Lemmas 4.3.25 through 4.3.29 below, we make use of the following definitions. Fix an arbitrary color ℓ, any ℓ-epoch h, and any two rounds i and j in h such that $i < j$ and algorithm *ΔLRU-EDF* evicts ℓ from the cache in the reconfiguration phase s of round i, and keeps ℓ out of the cache until the reconfiguration phase s' of round j, in which *ΔLRU-EDF* loads ℓ into the cache.

Lemma 4.3.25. *Let V be the sequence of rounds that starts with the round immediately following round i and ends with round j. The sequence V does not contain a round k such that k is an integral multiple of D_ℓ.*

Proof. Let V' be the sequence of rounds that starts with round i and ends with the round immediately preceding round j. It is sufficient to argue that V' does not contain any round k' such that $k' \bmod D_\ell = D_\ell - 1$. Suppose that V' contains such a round k'. Then in the drop phase of round k', all pending jobs of color ℓ are dropped, and ℓ becomes ineligible, at which point epoch h ends. This contradicts the fact that round j is also contained in h. Hence the assumption does not hold and the lemma follows. □

Corollary 4.3.1. *The deadline and timestamp of color ℓ do not change in phases s through s'.*

Proof. The corollary follows from the fact that the timestamp and deadline of ℓ can only increase in the arrival phase of a round k such that k is an integral multiple of D_ℓ and Lemma 4.3.25. □

Lemma 4.3.26. *Color ℓ is not selected by the LRU principle in phase s'.*

Proof. Let X be the set of colors selected by the LRU principle in phase s. Let ℓ' be any color in X. Since color ℓ is evicted from the cache in phase s, we have (1) $|X| = \frac{n}{4}$, (2) $\ell \notin X$, and (3) ℓ' precedes ℓ in the ordering maintained by the LRU principle in phase s. By Corollary 4.3.1, the timestamp of ℓ does not change in phases s through s'. Since the timestamp of a color does not decrease, ℓ' precedes ℓ in the ordering maintained by the LRU principle in phase s'. Since ℓ' is any color in X, and $|X| = \frac{n}{4}$, in phase s', color ℓ is not among the top $\frac{n}{4}$ positions of the ordering maintained by the LRU principle. $\square$

Lemma 4.3.27. *Color ℓ is nonidle in phases s through s'.*

Proof. By Lemma 4.3.26, in phase s', color ℓ is loaded into the cache by the EDF principle, which indicates that ℓ is nonidle. By Lemma 4.3.25, no color ℓ jobs arrive in phases s through s'. Hence, ℓ is nonidle in phases s through s'. $\square$

Lemma 4.3.28. *If we rank the colors in increasing order of deadlines with ties broken as in EDF, then the ranks of the colors are consistent in all reconfiguration phases.*

Proof. Since job arrivals are batched, for any color ℓ', the deadline of ℓ' increases by $D_{\ell'}$ in the arrival phase of each integral multiple of $D_{\ell'}$. The lemma then follows from the way that we break ties, and the fact that all delay bounds are powers of 2. $\square$

Lemma 4.3.29. *In round j, if loading color ℓ results in an EDF eviction, then the evicted color ℓ' is idle.*

Proof. Since $i < j$, $j \geq 1$. Let X be the set of colors selected by the EDF principle in the reconfiguration phase s'' of round $j-1$. By Lemma 4.3.27, in phase s'', ℓ' precedes ℓ in increasing order of deadlines (with ties broken as in *EDF*). By Lemma 4.3.28, in phase s', ℓ' precedes ℓ in increasing order of deadlines (with ties broken as in *EDF*). By Lemma 4.3.26, ℓ is loaded into the cache by the EDF principle in phase s'. Hence, ℓ is idle in phase s'. □

Proof of Lemma 4.3.7. We associate 4Δ units of credit with each epoch: 2Δ units of "first-time" credit and 2Δ units of "end-of-epoch" credit. We also associate 2Δ units of credit with each LRU eviction. Since there are at least Δ jobs of each color, by Lemma 4.3.5, the total credit is $O(\mathit{Cost}(\mathit{OFF}, \sigma, m) + \mathit{numEpochs}(\sigma) \cdot \Delta)$. It is sufficient to show that the total reconfiguration cost incurred by *ΔLRU-EDF* can be paid for by the credit.

Consider any color ℓ and any ℓ-epoch h. If *ΔLRU-EDF* does not load ℓ into the cache in h, then it does not incur any reconfiguration cost in h. Otherwise, let rounds $i_0 < \cdots < i_k$ be the rounds in h in which *ΔLRU-EDF* loads ℓ into the cache. For every j such that $0 \leq j \leq k$, let R_j be the reconfiguration operation performed by *ΔLRU-EDF* to bring in ℓ in round i_j. Since each cached color is replicated in *ΔLRU-EDF*, the cost of operation R_j is 2Δ. We use the 2Δ units of "first-time" credit associated with h to pay for operation R_0. In the following, we show that the remaining R_j's can also be paid for.

Consider any integer j such that $0 < j \leq k$. It is not hard to see that, when color ℓ is loaded into the cache in round i_j, some color ℓ' is evicted. If the eviction of color ℓ' is an LRU eviction, operation R_j can be paid for by the 2Δ units of credit associated with the LRU eviction. If the eviction of color ℓ' is an EDF eviction, then Lemma 4.3.29 implies that color ℓ' is evicted idle in round i_j. Since jobs of color ℓ' arrive only at integral multiples of $D_{\ell'}$, ℓ' remains idle until the next integral multiple of $D_{\ell'}$, at which point ℓ' becomes ineligible and its current ℓ-epoch h' ends. Hence, we can use the "end-of-epoch" credit associated with h' to pay for operation R_j. It is not difficult to argue that each unit of credit is used at most once. This completes the proof. □

4.4 Batched Arrivals

In this section, we solve $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, where each D_ℓ is a power of 2. This problem is characterized by a fixed reconfiguration cost Δ, a unit drop cost, per-color delay bounds D_ℓ, and batched arrivals (jobs of color ℓ arrive at integral multiples of D_ℓ).

As mentioned in Section 1, $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$ is a building block to solve our main problem $[\Delta \mid 1 \mid D_\ell \mid 1]$. To solve $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, we use a reduction to rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, which is solved in Section 4.3. Sections 4.4.1 and 4.4.2 give the reduction algorithm and analysis, respectively.

4.4.1 Algorithm *VarRecolor*

Consider an arbitrary request sequence σ for $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$. We define *recoloredReqSeq*(σ) as a request sequence obtained as follows. Let σ_i be request i of σ, where $0 \le i < |\sigma|$. For any color ℓ, we rank color ℓ jobs in σ_i in an arbitrary order. For any color ℓ and color ℓ job x in σ_i, we construct a job y that is the same as x except that the color of y is given by the pair (ℓ, j), where $j = \left\lfloor \frac{rank(x)}{D_\ell} \right\rfloor$, and $rank(x)$ is the rank of x in σ_i. Let σ'_i be the union of all such y's that are constructed over all colors ℓ. We obtain $\mathit{recoloredReqSeq}(\sigma)$ by concatenating the σ'_i's in increasing order of i.

Given any instance (σ, m) of $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, where each D_ℓ is a power of 2, algorithm *VarRecolor* proceeds as follows. Let $\sigma' = \mathit{recoloredReqSeq}(\sigma)$. First, we use algorithm *ΔLRU-EDF* on $(\sigma', 3m)$ to obtain an n-resource schedule S' for σ', where $n = O(m)$. Second, from S' we construct an n-resource schedule S for σ as follows. For any color ℓ, any integers j and k, whenever S' configures color (ℓ, j) on resource k, S configures color ℓ on resource k; whenever S' executes a job of color (ℓ, j) on resource k, S executes a job of color ℓ on resource k. Note that *VarRecolor* is an online algorithm.

4.4.2 Analysis of *VarRecolor*

In this section, we show that algorithm *VarRecolor* is resource competitive for $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, where each D_ℓ is a power of 2. First we establish some preliminary results.

Lemma 4.4.1. *For any request sequence σ for $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, where each D_ℓ is a power of 2, if there exists an m-resource schedule T for σ with cost C, then there exists a $3m$-resource schedule T' for $\mathit{recoloredReqSeq}(\sigma)$ with cost $O(C)$.*

Proof. See Section 4.4.3. $\square$

Lemma 4.4.2. *Consider any instance* (σ, m) *for* $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$*, where each* D_ℓ *is a power of* 2*. Let* $\sigma' = \mathit{recoloredReqSeq}(\sigma)$*. Let* S' *be the schedule produced by* Δ*LRU-EDF on* $(\sigma', 3m)$*. Let* S *be the schedule produced by VarRecolor on* (σ, m)*. Then* $\mathit{Cost}(S) \leq \mathit{Cost}(S')$*.*

Proof. By the definition of algorithm *VarRecolor*, S is obtained from S' by replacing color (ℓ, j), for any nonnegative integer j, with color ℓ. Hence, $\mathit{ReconfigCost}(S) \leq \mathit{ReconfigCost}(S')$. By the definition of $\mathit{recoloredReqSeq}(\sigma)$, the number of color ℓ jobs appearing in σ equals the total number of color (ℓ, j) jobs appearing in σ', over all nonnegative integers j. Hence, $\mathit{DropCost}(S) = \mathit{DropCost}(S')$. $\square$

Theorem 4.2. *Algorithm VarRecolor is resource competitive for* $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$*, where each* D_ℓ *is a power of* 2*.*

Proof. Consider any instance (σ, m) of $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$. Let T be the schedule produced by an arbitrary feasible offline algorithm on (σ, m). By the definition of a feasible algorithm, T uses m resources. Let $C = \mathit{Cost}(T)$ and $\sigma' = \mathit{recoloredReqSeq}(\sigma)$. By Lemma 4.4.1, there exists a $3m$-resource schedule T' for σ' with cost $O(C)$. Let S' be the schedule produced by algorithm *ΔLRU-EDF* on $(\sigma', 3m)$. By Theorem 4.1, S' uses $O(m)$ resources and incurs cost $O(C)$. Let S be the schedule generated by algorithm *VarRecolor* for (σ, m). By Lemma 4.4.2, S uses $O(m)$ resources and incurs cost $O(C)$. Hence, the theorem follows. $\square$

4.4.3 Offline to Offline Reduction

The purpose of this section is to provide proof of Lemma 4.4.1. We first give some definitions and preliminary results.

Consider any schedule S, any delay bound p, and any nonnegative integer i. Given an initial coloring μ, the coloring of the resources at the beginning of $block(p, i)$ is determined by S. As we mention in Section 2, if the initial coloring of S is not specified, we assume the default coloring in which resources are colored *black*. For any color ℓ, we define $Mono(S, p, i, \ell)$ as the set of resources k such that (1) the color of resource k at the beginning of $block(p, i)$ is ℓ, and (2) all jobs executed on resource k in $block(p, i)$, if any, are color ℓ jobs. We define $Mono(S, p, i)$ as $\cup_\ell Mono(S, p, i, \ell)$. We define $Multi(S, p, i)$ as the set of resources not in $Mono(S, p, i)$. We define $Full(S, p, i)$ (resp., $Empty(S, p, i)$) as the set of resources k such that each slot in resource k in $block(p, i)$ is occupied (resp., free) in S.

Let σ be any request sequence for $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, where each D_ℓ is a power of 2. Algorithm *Aggregate* takes an m-resource schedule S for σ and generates a $3m$-resource schedule. We initialize schedule T as a $3m$-resource schedule that drops all jobs in σ, and then modify T by assigning the jobs executed in S to execute in T. To modify T, we proceed in passes. In each pass we assign jobs with the next smallest delay bound p, block by block of delay bound p, in increasing order of block indices. We refer to the pass in which we assign the jobs with delay bound p as the *level-p pass*. For any delay bound p and any nonnegative integer i, we assign jobs in $block(p, i)$ in an arbitrary order of the colors with delay bound p.

For any delay bound p, any nonnegative integer i, and any color ℓ such that $D_\ell = p$, we assign color ℓ jobs in $block(p, i)$ in the following four phases. In the first phase, we label and configure the resources in $Mono(S, p, i, \ell)$ in T as follows. If $i = 0$, we label the resources in $Mono(S, p, i, \ell)$ from 0 to $|Mono(S, p, i, \ell)| - 1$ arbitrarily. Otherwise, for any resource k in the set $Mono(S, p, i, \ell) \cap Mono(S, p, i-1, \ell)$ such that the label assigned to resource k in $block(p, i-1)$ is in the range $[0, |Mono(S, p, i, \ell)|)$, we let resource k inherit its label in $block(p, i-1)$; we then assign the remaining labels in the range $[0, |Mono(S, p, i, \ell)|)$ to the other resources in $Mono(S, p, i, \ell)$ arbitrarily. We configure each resource k in $Mono(S, p, i, \ell)$ based on its current label as follows: In the reconfiguration phase of the first round of $block(p, i)$, configure resource k with color (ℓ, j), where j is the label assigned to resource k in $block(p, i)$.

In the second phase, we partition the set of color ℓ jobs executed in $block(p, i)$ in S into groups of size p. (One group may have size less than p.) In the next two phases, we are going to assign color ℓ groups to execute in the current schedule T.

The third phase proceed as follows. If $Mono(S, p, i, \ell) \cap Empty(T, p, i) = \emptyset$ or all color ℓ groups have been assigned, we terminate this phase. Otherwise, we proceed in the following three stages. In the first stage, among the color ℓ groups that have not been assigned, we pick a color ℓ group U of the largest size, breaking ties arbitrarily. In the second stage, we pick an arbitrary resource

k in $\mathit{Mono}(S, p, i, \ell) \cap \mathit{Empty}(T, p, i)$. In the third stage, we recolor group U as (ℓ, j), where j is the label we assign to resource k in $\mathit{block}(p, i)$ in the first phase, and then assign U to execute in the first $|U|$ slots in resource k.

The fourth phase proceeds as follows. If all color ℓ groups have been assigned, we terminate this phase. Otherwise, we proceed in the following five stages. In the first stage, we initialize q as $|\mathit{Mono}(S, p, i, \ell)|$. In the second stage, we pick an unassigned color ℓ group U of the largest size, breaking ties arbitrarily. In the third stage, we recolor U as (ℓ, q), and then increment q. In the fourth stage, we pick a resource k such that $k \in [m, 2m-1]$ and there are at least p free slots in $\mathit{block}(p, i)$ in resources k and $k+m$. (We will show such k exists in Lemma 4.4.5.) In the fifth stage, we assign U to execute in the first $|U|$ free slots in $\mathit{block}(p, i)$ in resources k and $k+m$.

Lemma 4.4.3. *For any request sequence σ for $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, where each D_ℓ is a power of 2, and any schedule S for σ, $T = \mathit{Aggregate}(S)$ is a schedule for $\sigma' = \mathit{recoloredReqSeq}(\sigma)$.*

Proof. The lemma follows from the definition of $\mathit{recoloredReqSeq}(\sigma)$, the way that we partition the jobs executed in S into groups, and the way that we recolor each group in algorithm *Aggregate*. □

Lemma 4.4.4. *Consider any request sequence σ for $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, where each D_ℓ is a power of 2, any schedule S for σ, any delay bound p, and the level-p pass of $\mathit{Aggregate}(S)$. For any nonnegative integer i, throughout the process* *in which we assign jobs in $block(p,i)$ in the level-p pass, all jobs executed in $block(p,i)$ in the current schedule are executed in $block(p,i)$ in S.*

Proof. Consider any point in time in the process in which we assign jobs in $block(p,i)$ in the level-p pass. Let T be the current schedule. Since we proceed in increasing order of delay bounds, all jobs executed in $block(p,i)$ in T have delay bounds at most p. From the way that we assign jobs, all jobs with delay bounds exactly p executed in $block(p,i)$ in T are executed in $block(p,i)$ in S. For each job x with delay bound q, $q < p$, that is executed in $block(p,i)$ in T, if x is executed in $block(q,j)$ in T, then x is executed in $block(q,j)$ in S. Since all delay bounds are power of 2, $block(q,j) \subset block(p,i)$. Hence, the lemma follows. $\square$

Lemma 4.4.5. *For any request sequence σ for $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, where each D_ℓ is a power of 2, and any schedule S for σ, the set of jobs executed by $T = Aggregate(S)$ equals that executed by S.*

Proof. It is sufficient to show that for any delay bound p and nonnegative integer i, throughout the process in which we assign jobs in $block(p,i)$ in the level-p pass, there exists a resource $k \in [m, 2m)$ such that there are at least p free slots in the resources k and $k+m$.

By Lemma 4.4.4, throughout the process in which we assign jobs in $block(p,i)$ in the level-p pass, all jobs executed in $[m, 3m)$ in $block(p,i)$ in the current schedule are executed in $block(p,i)$ in S. Since there are $2m$ resources with indices in the range $[m, 3m)$, throughout the process in which we assign jobs in $block(p,i)$ in the level-p pass, at least half the slots in the resources $[m, 3m)$ are free. Hence, the claim follows. $\square$

Lemma 4.4.6. *Consider any request sequence σ for $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, where each D_ℓ is a power of 2, and any schedule S for σ. Let $T = Aggregate(S)$. For any delay bound p, any nonnegative integer i, and any color ℓ such that $D_\ell = p$, the number of color (ℓ, j) jobs, over all nonnegative integers j, executed on the resources $[0, m)$ in $block(p,i)$ in T is at least the number of color ℓ jobs executed on $Mono(S, p, i, \ell)$ in $block(p,i)$ in S.*

Proof. From the way that we assign jobs, either $Mono(S, p, i, \ell) \subseteq Full(T, p, i)$ or each color ℓ job executed in $block(p,i)$ in S is recolored to color (ℓ, j), for some j, and is executed on the resources in $Mono(S, p, i, \ell)$ in T. Hence, the lemma follows. $\square$

Lemma 4.4.7. *Consider any request sequence σ for $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, where each D_ℓ is a power of 2, and any schedule S for σ. Let $T = Aggregate(S)$. Then the cost incurred by the reconfigurations on the resources $[m, 3m)$ in T is $O(Cost(S))$.*

Proof. For the purpose of our analysis, we assign credit in each pass, and refer to the credit assigned in the level-p pass as level-p credit. Consider any delay bound p, any nonnegative integer i, and each color ℓ such that $D_\ell = p$. To each color ℓ group that is assigned to execute on the resources $[m, 3m)$ in $block(p,i)$ in T, we assign 4Δ units of credit: Δ units of "start" credit, Δ units of "end" credit, and 2Δ units of "wrap-around" credit. By Lemma 4.4.6, the number of color (ℓ, j) jobs, over all nonnegative integers j, executed on the resources $[0, m)$ in *block*(p, i) in T is at least the number of color ℓ jobs executed on *Mono*(S, p, i, ℓ) in *block*(p, i) in S. Since each color ℓ jobs executed in *block*(p, i) in S is recolored as (ℓ, j) for some j, and executed in *block*(p, i) in T, the number of color (ℓ, j) jobs, over all nonnegative integers j, executed on the resources $[m, 3m)$ in *block*(p, i) in T is at most the number of color ℓ jobs executed on the resources in *Multi*(S, p, i) in *block*(p, i) in S. Hence, the number of color ℓ groups assigned to execute on the resources $[m, 3m)$ in *block*(p, i) is bounded by the number of reconfigurations from or to color ℓ in *block*(p, i) in S. Summing up over all colors ℓ such that $D_\ell = p$, and all nonnegative integers i, the total level-p credit is within a constant factor of the reconfigurations from or to colors ℓ such that $D_\ell = p$. Summing up over all delay bounds p, the total credit is within a constant factor of the reconfiguration cost of S.

It remains to show that the cost incurred by the reconfigurations on the resources $[m, 3m)$ in T is at most the total credit. Consider any group U assigned to the resources $[m, 3m)$. For convenience of presentation, we sort the jobs in U in the same order as the order of slots to which they are assigned. We use Δ units of start credit assigned to U to pay for the reconfiguration incurred by the execution of the first job in U, if any. We use Δ units of end credit assigned to U to pay for the reconfiguration incurred by the job following the last job in U, if any. We use 2Δ units of wrap-around credit assigned to U to pay for the reconfigurations incurred if U is wrapped around when the boundary of $block(p, i,)$ is encountered when we assign the jobs in U. Note that, if the jobs in U are not executed contiguously for any reason other than wrap-around, U skips past the groups previously laid down, either in the current pass or in the previous pass. From the way that we use the credit, the reconfigurations incurred by skipping past groups are paid for by the credit assigned to the groups being skipped past. $\square$

Lemma 4.4.8. *Consider any request sequence σ for $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, where each D_ℓ is a power of 2, and any schedule S for σ. Let $T = Aggregate(S)$. Then the cost incurred by the reconfigurations on the resources $[0, m)$ in T is $O(Cost(S))$.*

Proof. Consider any delay bound p and any nonnegative integer i. We assign credit in $block(p, i)$ in the level-p pass as follows. Consider any color ℓ such that $D_\ell = p$. Since we assume that the initial coloring is the default coloring in which all resources are black, by definition, $Mono(S, p, 0, \ell) = \emptyset$. Hence, if $Mono(S, p, i, \ell) \neq \emptyset$, then $i > 0$. Consider any resource $k \in Mono(S, p, i, \ell)$. We assign Δ units of credit to resource k if one of the following conditions holds: (1) resource $k \in Multi(S, p, i-1)$, and (2) resource $k \in Mono(S, p, i-1, \ell)$, and the label of resource k in $block(p, i)$ is different from that in $block(p, i-1)$. From the way that we assign labels, the number of resources k in $Mono(S, p, i, \ell) \cap Mono(S, p, i-1, \ell)$ such that the labels assigned to resource k in $block(p, i)$ is different from that in $block(p, i-1)$ is bounded by $|Mono(S, p, i-1, \ell)| - |Mono(S, p, i, \ell)|$, which in turn is at most

$|Mono(S, p, i-1, \ell) \cap Multi(S, p, i)|$. Hence, the level-$p$ credit assigned to the resources in $Mono(S, p, i, \ell)$ in $block(p, i)$ is within a constant factor of the cost incurred by the reconfigurations to color ℓ in $block(p, i)$.

We define the level-p credit as the credit that we assign in the level-p pass. We define a level-p reconfiguration as a reconfiguration to a color ℓ such that $D_\ell = p$. We argue that each level-p reconfiguration in $block(p, i)$ in T can be paid for by the level-p credit assigned in $block(p, i)$. From the way that we assign jobs, level-p reconfigurations on the resources $[0, m)$ in $block(p, i)$ in T are only made in the first round of $block(p, i)$ on the resources in $\cup_{D_\ell = p} Mono(S, p, i, \ell)$. Consider any color ℓ such that $D_\ell = p$. Consider any resource k in $Mono(S, p, i, \ell)$. If $i = 0$, we have argued that $Mono(S, p, i, \ell) = \emptyset$. Otherwise, let j and j' denote the label that we assign to resource k in $block(p, i)$ and $block(p, i-1)$, respectively. There are three possible cases. In the first case, resource k is in $Mono(S, p, i-1, \ell)$ and $j = j'$. In this case, there is no level-p reconfiguration on resource k in $block(p, i)$ in T. In the second case, resource k is in $Mono(S, p, i-1, \ell)$ and $j \neq j'$. In the third case, resource k is in $Multi(S, p, i-1)$. In the second and third cases, from the way that we assign credit, resource k is assigned Δ units of level-p credit in $block(p, i)$, which can pay for the reconfiguration on resource k in $block(p, i)$. Note that, by the definition of $Mono(S, p, i, \ell)$, the initial color of resource k in $block(p, i)$ is ℓ, which indicates that resource k is not in $Mono(S, p, i-1, \ell')$, for any color ℓ' such that $\ell' \neq \ell$. Summing up over all colors such that $D_\ell = p$, the cost incurred by the level-p reconfigurations on the resources $[0, m)$ in $block(p, i)$ in

T is at most the total level-p credit assigned in $block(p, i)$.

Therefore, the cost incurred by the level-p reconfigurations on the resources $[0, m)$ in $block(p, i)$ in T is within a constant factor of the cost incurred by the reconfigurations to color ℓ in $block(p, i)$ in S. Summing up over all nonnegative integers i and all delay bounds p, the lemma follows. $\square$

Lemma 4.4.1 follows from Lemmas 4.4.3, 4.4.5, 4.4.7, and 4.4.8.

4.5 Main Problem

In this section, we solve our main problem $[\Delta \mid 1 \mid D_\ell \mid 1]$, which is characterized by a fixed reconfiguration cost Δ, a unit drop cost, per-color delay bounds D_ℓ, and nonbatched arrivals (requests can arrive at any round).

To simplify the presentation, we focus on the special case where each D_ℓ is a power of 2. This special case is solved by a reduction to $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, which is solved in Section 4.4. For any color ℓ such that D_ℓ is equal to 1, jobs of color ℓ are already batched. For convenience, we focus on the case where D_ℓ is greater than 1, for all colors ℓ. Sections 4.5.1 and 4.5.2 give the algorithm and analysis for the reduction, respectively. Section 4.5.3 comments on how to extend our solution to arbitrary delay bounds, that is, to delay bounds that are not necessarily powers of 2.

4.5.1 Algorithm *VarBatch*

Let σ be an arbitrary request sequence for $[\Delta \mid 1 \mid D_\ell \mid 1]$. We define *batchedReqSeq*(σ) as a request sequence obtained by moving the arrival of any job x of color ℓ that arrives in *halfBlock*(D_ℓ, i) in σ to the beginning of *halfBlock*$(D_\ell, i+1)$, and changing the delay bound of x to $\frac{D_\ell}{2}$. Thus, *batchedReqSeq*(σ) can be viewed as a request sequence for $[\Delta \mid 1 \mid \frac{D_\ell}{2} \mid \frac{D_\ell}{2}]$.

Algorithm *VarBatch* proceeds as follows. First, given an arbitrary instance (σ, m) of $[\Delta \mid 1 \mid D_\ell \mid 1]$, we construct an instance $(\sigma', 7m)$ of $[\Delta \mid 1 \mid \frac{D_\ell}{2} \mid \frac{D_\ell}{2}]$, where $\sigma' =$ *batchedReqSeq*(σ). Second, we apply algorithm *VarRecolor* (defined in Section 4.4.1) on $(\sigma', 7m)$ to obtain a schedule S' for σ'. Finally, we obtain a schedule S for σ from S'. The schedule S is the same as S' except that the request sequence associated with S is σ. Note that algorithm *VarBatch* is an online algorithm.

4.5.2 Analysis of *VarBatch*

In this section, we show that algorithm *VarBatch* is resource competitive for $[\Delta \mid 1 \mid D_\ell \mid 1]$, where each D_ℓ is a power of 2. First we give some definitions and preliminary results.

Consider any request sequence σ for $[\Delta \mid 1 \mid D_\ell \mid 1]$, and any schedule T for σ. For any color ℓ and any color ℓ job x that arrives in *halfBlock*(D_ℓ, i) in σ, we say that the execution of x in T is σ-*early* (resp., σ-*punctual*, σ-*late*) if x is executed in *halfBlock*(D_ℓ, i) (resp., *halfBlock*$(D_\ell, i+1)$, *halfBlock*$(D_\ell, i+2)$) in T.

Lemma 4.5.1. *For any request sequence σ for $[\Delta \mid 1 \mid D_\ell \mid 1]$, and any schedule T for σ, if all job executions in T are σ-punctual, then T is also a schedule for batchedReqSeq(σ).*

Proof. The lemma follows immediately from the definition of a σ-punctual execution and the definition of *batchedReqSeq*(σ). □

Lemma 4.5.2. *For any request sequence σ for $[\Delta \mid 1 \mid D_\ell \mid 1]$, if there exists an m-resource schedule T for σ such that all job executions in T are σ-early, then there exists a $3m$-resource schedule T' for σ such that all job executions in T' are σ-punctual, and Cost$(T') = O($Cost$(T))$.*

Proof. The proof proceeds in two phases. In the first phase, we describe a procedure that constructs a $3m$-resource schedule T' in which all job executions are σ-punctual. In the second phase, we show that *Cost*$(T') = O($*Cost*$(T))$.

The first phase proceeds as follows. We start with a schedule that drops all jobs that appear in σ, and then modify the schedule by assigning the jobs executed in T to execute in the current schedule as follows. Consider any integer k such that $0 \leq k < m$. Let X_k be the set of jobs that are executed on resource k in T. We say that a job x in X_k is *k-special* if in schedule T, resource k is configured with the color of x, call it ℓ, throughout *halfBlock*(D_ℓ, i) and *halfBlock*$(D_\ell, i+1)$, and x is executed on resource k in *halfBlock*(D_ℓ, i). Any job in X_k that is not k-special is said to be *k-regular*. We assign k-special jobs to execute on resource $3k$ in the following manner: For any color ℓ and any k-special job x of color ℓ that is executed in round j in T, we assign x to execute in round $j + \frac{D_\ell}{2}$.

We assign k-regular jobs to execute on resources $3k+1$ and $3k+2$. To avoid collisions (i.e., different jobs being executed on the same resource and in the same round), we proceed in the following manner. We assign k-regular jobs in increasing order of delay bounds. For any delay bound p and any nonnegative integer i, let $V_{p,i,k}$ be the set of k-regular jobs with delay bound p that are executed in $\mathit{halfBlock}(p,i)$ in T. For any color ℓ with delay bound p, let $V_{p,i,k,\ell}$ be the color ℓ jobs in $V_{p,i,k}$. To assign the jobs in $V_{p,i,k}$, we iteratively consider each color ℓ such that $D_\ell = p$, in an arbitrary order, and assign the jobs in $V_{p,i,k,\ell}$ to the first $|V_{p,i,k,\ell}|$ free slots in $\mathit{halfBlock}(p,i+1)$ on resources $3k+1$ and $3k+2$. Let T' denote the schedule obtained by the procedure described above. It is not hard to see that all job executions in T' are σ-punctual.

In the second phase, we show that $\mathit{Cost}(T') = O(\mathit{Cost}(T))$. Consider any integer k, where $0 \le k < m$. Let C_k be the reconfiguration cost incurred on resource k in T. It is sufficient to show the following two claims: (1) all jobs in X_k are executed in T'. (2) the reconfiguration cost incurred by T' associated with the jobs in X_k is $O(C_k)$. Recall that X_k is the set of jobs executed on resource k in T, and assigned to execute on resources $3k$ through $3k+2$.

To argue claim (1), it is sufficient to show that there are no collisions as we assign the jobs in X_k. It is straightforward to argue that assigning the k-special jobs does not incur any collisions. It remains to argue that assigning the k-regular jobs does not incur any collisions. Consider any delay bound p and any nonnegative integer i, and the process in which we assign the jobs in $V_{p,i,k}$ (i.e., the set of k-regular jobs with delay bound p that are executed on resource k in *halfBlock*(p, i) in T) to execute in *halfBlock*$(p, i+1)$. Since we assign jobs in increasing order of delay bounds, and all delay bounds are powers of 2, all jobs assigned to execute in *halfBlock*$(p, i+1)$ before or during this process are executed either in *halfBlock*$(p, i+1)$ or in *halfBlock*(p, i) in T. Hence, with two resources (i.e., resources $3k+1$ and $3k+2$), we do not incur any collisions during this process.

In the remainder of this proof, we argue claim (2). It is straightforward to show that the reconfiguration cost incurred by T' associated with the k-special jobs is at most C_k. It remains to account for the reconfiguration cost incurred by T' associated with the k-regular jobs. We refer to each $V_{p,i,k,\ell}$, for any delay bound p, any nonnegative integer i, and any color ℓ, as a *k-group*. We assign credit as follows. Consider any reconfiguration operation R from color ℓ to color ℓ' on resource k in schedule T. Let $p = D_\ell$ and $q = D_{\ell'}$. Suppose R occurs in *halfBlock*(p, i) and in *halfBlock*(q, j). We assign 4Δ units of credit to each of the following k-groups: (a) $V_{q,j,k,\ell'}$, (b) $V_{p,i,k,\ell}$, and (c) if $i > 0$, $V_{p,i-1,k,\ell}$. It is not hard to see that the total credit is $O(C_k)$, and each k-group is assigned at least 4Δ units of credit.

Consider any k-group U. For the purpose of our analysis, we sort the jobs in U in the same order as the order of the slots to which they are assigned. We use Δ units of the credit assigned to U to pay for the reconfiguration incurred by the first job in U, if any. We use Δ unit of the credit assigned to U to pay for the reconfiguration incurred by the job following the last job in U, if any. We use 2Δ units of credit assigned to U to pay for the reconfigurations incurred if U is wrapped around when the boundary of the relevant block is encountered as we assign the jobs in U. Note that, if the jobs in U are not executed contiguously for any reason other than wrap-around, U skips past the k-groups previously laid down. From the way that we allocate the credit, the reconfigurations incurred by skipping past groups are paid for by the credit assigned to the groups being skipped past. Hence, all reconfigurations incurred by T' associated with k-regular jobs can be paid for by the total credit. $\square$

We omit the the proof of the following lemma since it is analogous to the proof of Lemma 4.5.2.

Lemma 4.5.3. *For any request sequence σ for $[\Delta \mid 1 \mid D_\ell \mid 1]$, if there exists an m-resource schedule T for σ such that all job executions in T are σ-late, then there exists a $3m$-resource schedule T' for σ such that all job executions in T' are σ-punctual, and $Cost(T') = O(Cost(T))$.*

Lemma 4.5.4. *For any request sequence σ for $[\Delta \mid 1 \mid D_\ell \mid 1]$, if there exists an m-resource schedule T for σ, then there exists a $7m$-resource schedule T' for batchedReqSeq(σ) such that $Cost(T') = O(Cost(T))$.*

Proof. We break each request in σ into three requests: one consisting of the jobs for which the executions are σ-early in T, one consisting of the jobs for which the executions are σ-punctual in T, and one consisting of the jobs for which the executions are σ-late in T. Let α (resp., β, γ) denote the resulting request sequence involving the jobs for which the executions are σ-early (resp., σ-punctual, and σ-late) in T. We define T_{early} (resp., T_{punct}, T_{late}) as the schedule obtained by removing the jobs in β and γ (resp., α and γ, α and β) from T. Let $C = Cost(T)$. It is not hard to see that $Cost(T_{early})$ (resp., $Cost(T_{punct})$, $Cost(T_{late})$) is at most $O(C)$.

By Lemma 4.5.2, there exists a $3m$-resource schedule T'_{early} for α such that all job executions in T'_{early} are α-punctual and $Cost(T'_{early}) = O(C)$. By Lemma 4.5.3, there exists a $3m$-resource schedule T'_{late} for γ such that all job executions in T'_{late} are γ-punctual and $Cost(T'_{late}) = O(C)$.

We construct a $7m$-resource schedule T' as follows. On resources 0 through $m-1$, T' behaves the same as T_{punct}. On resources m through $4m-1$ resources, T' behaves the same as T'_{early}. On resources $4m$ through $7m-1$, T' behaves the same as T'_{late}. It is not hard to see that T' is a schedule for σ, all jobs in T' are σ-punctual, and $Cost(T') = O(C)$. The lemma then follows from Lemma 4.5.1. $\square$

Theorem 4.3. *Algorithm VarBatch is resource competitive for* $[\Delta \mid 1 \mid D_\ell \mid 1]$, *where each* D_ℓ *is a power of* 2.

Proof. Consider an arbitrary instance (σ, m) of $[\Delta \mid 1 \mid D_\ell \mid 1]$. Let T be a schedule produced by any feasible offline algorithm on (σ, m). By the definition of a feasible algorithm, T uses m resources. Let $C = Cost(T)$, and $\sigma' =$

batchedReqSeq(σ). By Lemma 4.5.4, there exists a $7m$-resource schedule T' for σ' with cost $O(C)$.

Since σ' can be viewed as a request sequence for $[\Delta \mid 1 \mid \frac{D_\ell}{2} \mid \frac{D_\ell}{2}]$, Theorem 4.2 implies that the schedule S' produced by *VarRecolor* on $(\sigma', 7m)$ uses $O(m)$ resources and incurs cost $O(C)$. By the definition of algorithm *VarBatch*, the schedule S produced by *VarBatch* on (σ, m) is the same as S' except that the associated request sequence is σ. Hence, the lemma follows. □

4.5.3 Extension to Arbitrary Delay Bounds

The extension of our solution to arbitrary delay bounds is straightforward. The basic idea is as follows: For any delay bound p such that $2^j \leq p < 2^{j+1}$, and any job x with delay bound p that arrives in *halfBlock*$(2^j, i)$, we delay the arrival of x to the beginning of *halfBlock*$(2^j, i+1)$, and change the delay bound of x to 2^{j-1}. The proof that the extended solution is resource competitive is similar to the proof given in Section 4.5.2.

Chapter 5

Concluding Remarks

In this dissertation, we initiate the study of the class of reconfigurable resource scheduling problems within the framework of competitive analysis. We study a subclass in this broad class, and provide resource competitive online algorithms for two main problems in this subclass, namely, reconfigurable resource scheduling with variable drop costs, and reconfigurable resource scheduling with variable delay bounds. In solving both problems, we adopt a layered approach where in each layer we reduce to a scheduling problem defined over a more constrained set of possible inputs, and thereby simplify the problem.

In solving reconfigurable resource scheduling with variable drop costs, we are able to reduce the main problem to a caching problem, which we refer to as file caching with remote reads. This caching problem generalizes the file caching problem, and is a special case of the k-server problem with excursions, and may be of independent interest. Our solution to reconfigurable resource scheduling with variable delay bounds is based on a natural and novel combination of the EDF and LRU principles.

In the subclass of reconfigurable resource scheduling problems considered in this dissertation, there are variable problem dimensions: variable job execution time, variable reconfiguration costs, variable drop costs, and variable delay bounds. In each of the two main problems that we solve in this dissertation, we allow one of problem parameters to vary arbitrarily, and fix other problem parameters. It is interesting to investigate the existence of resource competitive online algorithms handling other combinations of the problem dimensions. In particular, it would be interesting to see that whether one can extend our *ΔLRU-EDF* algorithm to solve more general variants.

Throughout this dissertation, we associate an explicit cost with the reconfiguration of a resource. An alternative is to consider that the reconfiguration incurs a context switch time during which the resource cannot process any jobs. Problems of this sort have been studied in the offline setting [6, 7] (see [5, Chapter 9] for a survey) and in experimental work [16, 17]. Yet, within the framework of competitive analysis, such problems remain largely unexplored. It would be interesting to see whether some of the techniques used in this dissertation can be applied to reconfigurable resource scheduling problems that involve context switch time.

Appendices

Appendix A

Analysis of *EDF*

In this section, we show that *EDF* (defined in Section 4.3.2) is not constant competitive, even if *EDF* is given an arbitrary constant factor resource advantage, and an arbitrary constant replication factor r, that is, if each color in the cache is replicated in r locations. Algorithm *EDF* is a specific algorithm defined using the EDF principle. We expect that other algorithms based on the EDF principle are also subject to similar lower bounds.

Consider an arbitrary instance (σ, m) of rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$. Let *OFF* denote an arbitrary feasible offline algorithm. We assume that n, the number of resources that *EDF* can use, is equal to rsm, where r is the replication factor, and s is an arbitrary positive constant. We assume $(s+1)\cdot m$ colors as follows: m colors with delay bound 2^j, m colors with delay bound 2^k, m colors with delay bound 2^{k+1}, $\ldots$, and m colors with a delay bound 2^{k+s-1}, where $2^k > 2^j > \Delta$. We refer to each color with delay bound 2^j as a short-term color, and we refer to each of the other colors as a long-term color. The request sequence proceeds in 2^{k+s-1} rounds as follows. For each short-term color, we receive Δ jobs at each integral multiple of 2^j, in rounds 0 through $2^{k-1} - 1$. For each long-term color with a delay bound of 2^{k+i}, for

$0 \leq i < s$, we receive 2^{k+i-1} jobs in round 0.

Consider rounds 0 through $\frac{2^{k-1}}{r}$. Each long-term color always has jobs to execute. Since $2^j > \Delta$, each short-term color is brought into the cache and then evicted $\frac{2^{k-1}}{2^j r}$ times. Hence, the reconfiguration cost incurred by *EDF* is $\Omega(2^{k-j} m\Delta)$.

Suppose that *OFF* caches the short-term colors in rounds 0 through $2^{k-1}-1$, and caches the colors with delay bound 2^{k+i} in rounds 2^{k+i-1} through $2^{k+i}-1$, where $0 \leq i < s$. Algorithm *OFF* does not incur any drop cost and incurs a reconfiguration cost of $O(m\Delta)$. Hence the competitive ratio of *EDF* is $\Omega(2^{k-j})$, which can be made arbitrarily large by setting j and k appropriately.

Appendix B

Analysis of ΔLRU

In this section, we show that ΔLRU (defined in Section 4.3.3) is not constant competitive, even if ΔLRU is given an arbitrary constant factor resource advantage, and an arbitrary constant replication factor r, that is, if each color in the cache is replicated in r locations. Algorithm ΔLRU is a specific algorithm defined using the LRU principle. We expect that other algorithms based on the LRU principle are also subject to similar lower bounds.

Consider an arbitrary instance (σ, m) of rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$. Let OFF denote an arbitrary feasible offline algorithm. We assume that n, the number of resources that ΔLRU can use, is equal to rsm, where r is the replication factor, and s is an arbitrary positive constant. Consider sm colors with delay bound 2^j and m colors with delay bound 2^k, where $2^k > 2^j > \Delta$. We refer to each color with delay bound 2^j as a short-term color, and we refer to each color with delay bound 2^k as a long-term color. The request sequence proceeds in 2^k rounds as follows. We receive Δ jobs of each short-term color at each integral multiple of 2^j, and 2^k jobs of each long-term color in round 0.

It is not hard to verify that, from the reconfiguration phase of round 2^j, the timestamp of any short-term color is more recent than that of any long-term color. Hence, in the reconfiguration phase of round 2^j, $\mathit{\Delta LRU}$ caches all short-term colors, and evicts all long-term colors. After round 2^j, $\mathit{\Delta LRU}$ does not change the configuration. Thus, the drop cost incurred by $\mathit{\Delta LRU}$ is at least $(2^k - 2^j)m$. Since $k > j$, the cost incurred by $\mathit{\Delta LRU}$ is $\Omega(2^k m)$.

Suppose that *OFF* caches the long-term colors throughout. The reconfiguration cost incurred by *OFF* is $m\Delta$. The drop cost incurred by *OFF* is $2^{k-j} sm\Delta$. Hence the total cost incurred by *OFF* is $O(2^{k-j} m\Delta)$. Thus, the competitive ratio of $\mathit{\Delta LRU}$ is $\Omega(\frac{2^k m}{2^{k-j} m\Delta}) = \Omega(\frac{2^j}{\Delta})$, which can be made arbitrarily large by setting j and k appropriately.

Bibliography

[1] S. Albers and H. Koga. Page migration with limited local memory capacity. In *Workshop on Algorithms and Data Structures*, pages 147–158, August 1995.

[2] Y. Bartal, M. Charikar, and P. Indyk. On page migration and other relaxed task systems. In *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 43–52, January 1997.

[3] Y. Bartal, S. Leonardi, A. Marchetti-Spaccamela, J. Sgall, and L. Stougie. Multiprocessor scheduling with rejections. In *Proceedings of the 7th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 95–113, January 1996.

[4] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, Cambridge, 1998.

[5] P. Brucker. *Scheduling Algorithms*. Springer-Verlag, Berlin, 2001.

[6] P. Brucker, M. Y. Kovalyov, Y. M. Shafransky, and F. Werner. Batch scheduling with deadlines on parallel machines. *Annals of Operations Research*, 83:23–40, 1998.

[7] J. Bruno and P. Downey. Complexity of task sequencing with deadlines, set-up times and changeover costs. *SIAM Journal on Computing*, 7:393–403, 1978.

[8] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems*, pages 193–206, December 1997.

[9] A. Chandra, W. Gong, and P. Shenoy. Dynamic resource allocation for shared data centers using online measurements. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, pages 300–301, June 2003.

[10] J. S. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing energy and server resources in hosting centers. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 103–116, October 2001.

[11] M. Dertouzos. Control robotics: The procedural control of physical processors. In *Proceedings of the IFIP Congress*, pages 807–813, 1974.

[12] S. Irani. Page replacement with multi-size pages and applications to web caching. In *Proceedings of the 29th Symposium on the Theory of Computing*, pages 701–710, May 1997.

[13] S. Irani, S. Shukla, and R. Gupta. Algorithms for power savings. In *Proceedings of the 14th Symposium on Discrete Algorithms*, pages 37–46,

January 2003.

[14] S. Irani, S. Shukla, and R. Gupta. Online strategies for dynamic power management in systems with multiple power saving states. *ACM Transactions on Embedded Computing Systems*, 2:325–346, 2003.

[15] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM*, 47:617–643, 2000.

[16] R. Kokku. *ShaRE: Run-time System for High-performance Virtualized Routers*. PhD thesis, Department of Computer Science, University of Texas at Austin, August 2005.

[17] R. Kokku, T. Riché, A. Kunze, J. Mudigonda, J. Jason, and H. Vin. A case for run-time adaptation in packet processing systems. *ACM SIGCOMM Computer Communication Review*, 34:107–112, 2004.

[18] C. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20:46–61, 1973.

[19] M. S. Manasse, L. A. McGeoch, and D. D. Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11:208–230, 1990.

[20] N. Megiddo and D. S. Modha. Arc: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, pages 115–130, 2003.

[21] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The *LRU-K* page replacement algorithm for database disk buffering. In *Proceedings of ACM SIGMOD*, pages 297–306, May 1993.

[22] C. A. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. *Algorithmica*, pages 163–200, 2002.

[23] C. G. Plaxton, Y. Sun, M. Tiwari, and H. Vin. Reconfigurable resource scheduling. In *Proceedings of 18th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 93–102, July 2006.

[24] C. G. Plaxton, Y. Sun, M. Tiwari, and H. Vin. Reconfigurable resource scheduling with variable delay bounds. In *Proceedings of 22nd IEEE International Parallel and Distributed Processing Symposium*, pages 1–10, March 2007.

[25] D. Ramanathan, S. Irani, and R. Gupta. Latency effects of system level power management algorithms. In *Proceedings of the 2000 IEEE/ACM international conference on Computer-aided design*, pages 350–356, 2000.

[26] S. S. Seiden. More multiprocessor scheduling with rejection. Technical Report Woe–16, TU Graz, 1997.

[27] S. S. Seiden. *Randomization in On-line Computation.* PhD thesis, University of California, Irvine, 1997.

[28] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.

[29] T. Spalink, S. Karlin, L. L. Peterson, and Y. Gottlieb. Building a robust software-based router using network processors. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 216–229, October 2001.

[30] A. Srinivasan, P. Holman, J. Anderson, S. K. Baruah, and J. Kaur. Multiprocessor scheduling in processor-based router platforms: Issues and ideas. In *Proceedings of the 2nd Workshop on Network Processors*, pages 48–62, February 2003.

[31] J. S. Turner. New directions in communications (or which way to the information age?). *IEEE Communications Magazine*, 24:8–15, October 1986.

[32] H. Vin, J. Mudigonda, J. Jason, E. J. Johnson, R. Ju, A. Kunze, and R. Lian. A programming environment for packet-processing systems: Design considerations. In *Proceedings of the 3rd Workshop on Network Processors and Applications*, February 2004.

[33] N. E. Young. On-line file caching. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 82–86, January 1998.

Vita

Yu Sun was born in Jiande, Zhejiang, China. She attended Jiaxin No.1 middle school from 1992 to 1996. She received her B.E. degree in Computer Science and Engineering from Zhejiang University, Hangzhou, China, in June 2000. She received her M.S. degree in Computer Science from the University of Texas at Austin in December 2003.

Permanent address: WenChang Road, WenNanLi
Building 14, Room 104
JiaXing, Zhejiang Province, P.R.China 314001

This dissertation was typeset with LaTeX† by the author.

†LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.