Reconfigurable Resource Scheduling with Variable Delay Bounds

C. Greg Plaxton[1], Yu Sun[2], Mitul Tiwari[2], and Harrick Vin[2]
Department of Computer Science
University of Texas at Austin
{plaxton, sunyu, mitult, vin}@cs.utexas.edu

Abstract

Certain emerging network applications involve dynamically allocating shared resources to a variety of services to provide QoS guarantees for each service. Motivated by such applications, we address the following online scheduling problem belonging to the recently introduced class of reconfigurable resource scheduling: unit jobs of different categories arrive over time and need to be completed within category-specific delay guarantees, or else they are dropped at a unit drop cost; processors can be reconfigured to process jobs of a certain category at a fixed reconfiguration cost; the goal is to minimize the total cost. We study this problem in the framework of competitive analysis. Through a novel combination of the EDF and LRU scheduling principles, we obtain an online algorithm that is constant competitive when given a constant factor advantage in the number of resources over an optimal offline algorithm.

[1]Supported by NSF Grants CCR–0310970 and ANI–0326001.
[2]Supported by NSF Grant ANI–0326001 and Texas Advanced Technology Program Grant 003658-0608-2003.

1 Introduction

Motivation. Reconfigurable resource scheduling, recently introduced in [14], is a class of scheduling problems with the following salient features: (1) there are jobs of different categories; (2) resources can be reconfigured to process jobs of a certain category at an overhead, in terms of cost or time.

This paradigm is useful in multi-core and multi-processor environments that 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 (or support multiple categories of jobs) simultaneously (e. g., a shared data center and a multi-service router). To isolate — with respect to security and performance — categories from one another, these environments often configure processors to support only one category at a time. The set of processors configured to support a particular category depends upon the workload demands for that category; fluctuations in workloads require changes in processor allocations. For instance, a shared data center [4, 5] adjusts dynamically the allocation of processors to independent categories as the workload composition changes. Similarly, a multi-service router based on programmable, multi-core network processors [16, 17, 18] adjusts allocations of processors to different packet categories as the traffic load fluctuates. 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 [9].

Problem Statement. In this paper, we study and solve the following variant of reconfigurable resource scheduling. The input is a sequence of requests, each of which is a set of unit jobs. Each job has a category, and needs to be executed within a *category-specific* delay bound from its arrival, or else it is dropped at a unit drop cost. A job of a given category can only be executed on a resource configured for that category. A resource can be reconfigured at any time at a fixed reconfiguration cost. The objective is to minimize the total cost.

Our goal is to design online algorithms that provide good performance under all possible operating conditions. This motivates us to study this problem in the framework of competitive analysis, where the performance of an online algorithm is measured by the competitive ratio [15], that is, the maximum ratio between the cost incurred by the online algorithm and that incurred by an optimal offline algorithm, over all input sequences (see [1] for a comprehensive introduction to competitive analysis). In this paper, we adopt a standard technique in competitive analysis, sometimes referred to as *resource augmentation* [7, 13], in which the online algorithm is given extra resources as a method to compensate 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.

We aim to provide a resource competitive online algorithm for reconfigurable resource scheduling with variable delay bounds. In our previous work [14], we solve a variant with uniform delay bounds and variable drop costs. It is not clear whether the techniques used in that work can be generalized to solve the variant studied in this paper. 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 end up incurring a large number of reconfigurations, or dropping a lot of short-term jobs; later on, we may regret incurring these costs if we encounter a lengthy interval in which no short-term jobs arrive, and all of the background jobs could be executed using one 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 period, then later on, we may regret doing so if we never encounter a long idle interval. In summary, either of these basic approaches lead to *thrashing* (i. e., excessively high reconfiguration cost) or *underutilization* (i. e., excessively high drop cost). Even under resource augmentation, these basic approaches fail, when there are many categories of jobs with different delay bounds.

One natural approach 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 this setting. We have investigated various natural alternatives (See Section 3.1.1 for an example). In all of these alternatives, we encounter the following basic difficulty: if we configure the categories with the most recent LRU timestamp, without considering whether these categories have jobs to execute, then we may suffer from underutilization; if we configure the categories that have the most recent LRU timestamps, and *have jobs to execute*, then we may suffer from thrashing. Thus it appears that LRU alone is insufficient to obtain a resource competitive solution.

Another natural approach is to consider algorithms based on the Earliest Deadline First (EDF) principle. As with LRU, there are different ways we can formulate a specific algorithm based on the EDF principle (See Section 3.1.2 for an example). However, all EDF variations seem to suffer from thrashing, and therefore fail to yield a resource competitive solution.

Our Contribution. In this paper, we give a resource competitive online algorithm for reconfigurable resource

scheduling with variable delay bounds. We solve the problem using a layered approach. First, we use a batching subroutine to reduce the problem to the special case in which jobs of a given category arrive at integral multiples of the category-specific delay bound. This layer is analogous to the first layer in [14], but is more involved with variable delay bounds. Second, we reduce the batched problem to a rate-limited problem in which at most p jobs of category p arrive at each integeral multiple of p. Third, we solve the core problem using a novel combination of EDF and LRU, which we view as a major contribution of this paper. The main idea of this combination is to keep two sets of categories configured: one set consists of categories with the most recent timestamps, the other set consists of categories that have the earliest deadlines and have jobs to execute.

Our consideration of the EDF and LRU combination is motivated by the following observations. The LRU component, which does not consider idleness, allows categories with short delay bounds to remain cached as long as they have recent timestamps; this reduces thrashing. The EDF component ensures that the resources are well utilized. The main challenge in the analysis is to bound the reconfiguration cost. We address the challenge by employing amortized analysis and a proof technique called "phase partition" used in [15].

Related Work. In our previous work [14], we introduce the class of reconfigurable resource scheduling, and solve a variant with uniform delay bounds and variable drop costs by reducing to a file caching problem.

Brucker [2, Chapter 9] surveys a class of offline scheduling problems with context switch time, which they call changeover time. In this class of problems, 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 period, 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 time, Brucker et al. [3] 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 position paper, Srinivasan et al. [17] discuss scheduling problems for multi-core network processors, and consider the application of existing multiprocessor scheduling algorithms in this domain. Various challenges are identified and some initial ideas to address these concerns are presented. Kokku et al. [8] give a scheduling algorithm, called Everest, for multi-core network processors. The parameters considered are a per-service delay bound, a per-service execution requirement, and a fixed context switch time. Everest is shown to perform well in experiments in terms of maximizing the number of packets processed within a service-specific delay tolerance.

The EDF scheduling algorithm is shown [6, 10] to be an optimal preemptive uniprocessor scheduling algorithm for problems that do not involve reconfiguration overhead, in terms of the number of jobs executed. In this paper, we discuss the issues of applying EDF in reconfigurable resource scheduling with variable delay bounds, and propose a combination of EDF and LRU to address the problems.

The classic disk paging problem studied by Sleator and Tarjan [15] can be viewed as a special case of reconfigurable resource scheduling with unit delay bound, unit reconfiguration cost, infinite drop cost, and where each request consists of a single job. In this seminal work, the competitive ratio of any deterministic online paging algorithm is shown to be at least the cache size, and certain algorithms such as LRU are shown to be resource competitive.

O'Neil et al. [12] consider a variation of LRU called LRU-K, which keeps track of the times of the last K references to pages. Megiddo et al. [11] 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 deadline aspects by keeping two sets of categories configured: one set captures the recency aspect and the other captures the deadline aspect.

2 Preliminaries

Problem Definitions. Most of the material in this section also appears in the preliminaries section of [14]. We include it here in order to make the current presentation self-contained.

For the reconfigurable resource scheduling problems considered in this paper, the input is a sequence of requests, each of which consists of a (possibly empty) set of unit jobs. Each job is characterized by a non-black color, a nonnegative integer arrival time, and a positive integer delay bound. For any job, we define an associated deadline to be its arrival time plus its delay bound. A job has to be executed on a resource of the same color between its arrival time and its deadline, or else it is dropped at a unit drop cost. After a job arrives, it is *pending* until it is either dropped or executed.

There is a finite set of resources on which jobs are executed. Resources are numbered from 0. Each resource is associated with a color and can be reconfigured to a different color at any time at a fixed reconfiguration cost. Initially,

all resources are colored black.

Any problem considered here proceeds in rounds numbered from 0. Each round i consists of four phases, in the following order: (1) in the *drop phase*, jobs with deadline i are dropped; (2) in the *arrival phase*, the ith request is received; (3) in the *reconfiguration phase*, for each resource, an algorithm decides whether to reconfigure to a different color or not, and if so, to which color; (4) in the *execution phase*, for each resource configured to color ℓ, we execute up to one pending job of color ℓ.

For any request sequence σ, a *schedule* specifies the reconfigurations, if any, and the job executions, to perform in each round. The total cost of a schedule is the sum of all reconfiguration and drop costs. The goal is to device a schedule of minimum cost for a given request sequence σ.

Let S and S' be any two schedules for a given request sequence σ. We say S is *resource competitive* with S' if, the number of resources given to S is within a constant factor of that given to S', and the cost incurred by S is within a constant factor of that incurred by S'.

An offline algorithm knows all requests in advance. An online algorithm has to make decisions without knowing the future requests. The competitive ratio of an algorithm A is defined as the maximum ratio, over all request sequences σ, of the cost incurred by A on σ to that incurred by an optimal offline algorithm for σ. An algorithm A is defined to be c-competitive if the competitive ratio is c. Any c-competitive algorithm A is called constant competitive if c is a constant. We say an algorithm A is *resource competitive* if, for any request sequence σ, the schedule generated by A is resource competitive with an optimal schedule for σ.

For the sake of brevity, we use the $[reconfig \mid drop \mid delay \mid batch]$ notation introduced in [14]. The $reconfig$ field describes the details of the reconfiguration cost. In this paper, there is only one possible value for this field, a fixed reconfiguration cost denoted by Δ. The $drop$ field describes the details of the drop cost. In this paper, there is only one possible value for this field, namely 1, since we assume unit drop cost. The $delay$ field contains the details of the delay bound. In this paper, there is only one possible value for this field, a per-color delay bound denoted by 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 paper, the possible values for this field are 1 and D_ℓ.

With this notation, our main problem is denoted by $[\Delta \mid 1 \mid D_\ell \mid 1]$. The special case in which jobs of color ℓ arrive at integral multiples of D_ℓ is denoted by $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$. We use the terminology "rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$" to denote the special case of $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$ in which at most D_ℓ color ℓ jobs arrive at each integral multiple of D_ℓ. In this paper, we assume Δ is a positive integer (it is not hard to generalize our results to arbitrary Δ).

Roadmap. The rest of the paper is organized as follows. Section 3 solves rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, where each D_ℓ is a power of 2. Section 4 solves $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, where each D_ℓ is a power of 2, with a reduction to rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$. Section 5 solves our main problem $[\Delta \mid 1 \mid D_\ell \mid 1]$ by a reduction to $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$.

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 configuration cost Δ, a unit drop cost, per-color delay bound D_ℓ, batched arrivals (jobs with delay bound D_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).

To solve this problem, we propose a combination of EDF and LRU, referred to as algorithm ΔLRU-EDF. The main result of this section is presented in Theorem 1, which is used in Theorem 2 in Section 4.

3.1 Algorithms

In this section, we introduce three online algorithms for rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, where D_ℓ is a power of 2, namely, ΔLRU, EDF, and ΔLRU-EDF. Because these algorithms only differ in the way the resources are reconfigured, we first present the common aspects, and then define the reconfiguration schemes of these algorithms in Section 3.1.1, Section 3.1.2, and Section 3.1.3, respectively. The reconfiguration scheme of algorithm ΔLRU is a variation of LRU. The reconfiguration scheme of algorithm EDF is based on the earliest deadline principle. Even though neither ΔLRU nor EDF is resource competitive, the study of these two algorithms motivates our consideration of ΔLRU-EDF, which is a certain combination of ΔLRU and EDF, and which we show to be resource competitive.

We use n to denote the number of resources given to the online algorithm. We consider the set of resources as a cache where resource i is viewed as location i. We view each color as a page. Each location can cache one color. We view reconfiguring resource i with color ℓ as caching color ℓ at location i.

For each color ℓ, we maintain a counter and a deadline, denoted by $\ell.cnt$ and $\ell.dd$, respectively. A color ℓ is *idle* if there are no pending color ℓ jobs, and *nonidle* otherwise. A color is either *eligible* or *ineligible*.

The common aspects of the three algorithms are as follows. Initially, the cache is empty, and all colors are ineligible. In each round k, the actions performed in the four phases are described as follows.

Drop phase For any color ℓ, if k is an integral multiple of D_ℓ, we drop all pending color ℓ jobs, and set color ℓ to be ineligible and $\ell.cnt$ to zero if color ℓ is eligible and not in the cache.

Arrival phase For any color ℓ, if k is an integral multiple of D_ℓ, we receive a request, which consists of a set of jobs X, and perform the following steps.

1. We set the deadline of ℓ to be $k + D_\ell$.
2. We increase $\ell.cnt$ by the number of color ℓ jobs in X.
3. If $\ell.cnt$ is at least Δ, we perform the following substeps.
   (a) We set $\ell.cnt$ to ($\ell.cnt$ mod Δ), which we refer to as a *counter wrapping event* of color ℓ.
   (b) If color ℓ is ineligible, we set color ℓ to be eligible.

Reconfiguration phase We use the first half locations of the cache capacity to cache distinct colors; the method used depends on the algorithm, see Sections 3.1.1 through 3.1.3. We use the remaining cache capacity to replicate the cache content of the first half locations, that is, we maintain the invariant that each cached color is cached in two locations.

Execution phase For each resource q, let ℓ be the color cached at location q. We execute one pending job of color ℓ.

3.1.1 Reconfiguration Scheme of ΔLRU

Consider any color ℓ. Let k be the most recent integral multiple of D_ℓ. We define the *timestamp* of ℓ to be the index of the latest round before round k in which a counter wrapping event of color ℓ occurs, and 0 if such a round does not exist.

The reconfiguration scheme of algorithm ΔLRU works as follows. We maintain the invariant that we keep $\frac{n}{2}$ eligible colors with the most recent timestamps in the cache, breaking ties arbitrarily.

Intuitively, like the classic LRU algorithm, ΔLRU intends to capture the recency aspect of the input sequence. Due to the difference between the reconfiguration and drop costs, we update the timestamp of each color roughly Δ job arrivals of that color. To avoid caching a color with a deadline far ahead too aggressively (which may not be desirable since we can use the slack to execute jobs of other colors with earlier deadlines first), for each color ℓ, we only consider the counter wrapping events of ℓ for which a subsequent integral multiple of D_ℓ has elapsed.

At a high level, ΔLRU may keep idle colors with recent timestamps, which results in low utilization of resources. We refer the readers to Appendix A for a detailed example that shows ΔLRU is not resource competitive.

3.1.2 Reconfiguration Scheme of EDF

The reconfiguration scheme of EDF works as follows. We rank the eligible colors first on idleness, where nonidle colors come first, and then in ascending order of deadlines, breaking ties by increasing delay bounds, and then by a consistent order of colors. We update the cache as follows. If a nonidle eligible color ℓ in the top $\frac{n}{2}$ rankings is not in the cache, we cache color ℓ. In case there is not enough room in the cache, we evict the color with the lowest rank.

At a high level, EDF suffers from thrashing: when a color ℓ becomes idle and nonidle alternatively, a nonidle color ℓ' with larger delay bound is repeatedly brought in and removed from the cache, and so EDF incurs a large number of reconfigurations. We may later regret paying these reconfiguration costs if a long period of time appears later that allows to execute all jobs of ℓ' with a single reconfiguration. We refer the readers to Appendix B for a detailed example that shows EDF is not resource competitive.

3.1.3 Reconfiguration Scheme of ΔLRU-EDF

As discussed in Section 3.1.1 and Section 3.1.2, an algorithm that captures only the recency aspect or only the deadline aspect in the input sequence is not resource competitive. This observation motivates us to think about algorithms that capture both the recency and deadline aspects. In the following, we introduce the reconfiguration scheme of ΔLRU-EDF, which is a combination of ΔLRU and EDF.

The reconfiguration scheme of ΔLRU-EDF is as follows. We first run the reconfiguration scheme of ΔLRU to cache the $\frac{n}{4}$ eligible colors with the most recent timestamps. At any instant, a color is called an LRU-color if it is cached by ΔLRU, and a non-LRU color otherwise. We then rank the non-LRU colors that are eligible, in the same way as we rank eligible colors in the reconfiguration scheme of EDF (see Section 3.1.2 for details). Let X be the set of nonidle and non-LRU colors in the top $\frac{n}{4}$ rankings but not in the cache. We bring all colors in X into the cache. In case there is not enough room in the cache, we repeatedly evict the non-ΔLRU color with the lowest rank until there is enough room.

3.2 Analysis of algorithm ΔLRU-EDF

Consider any input sequence σ. For any color ℓ and any color ℓ job x, we say x is *ineligible* if x is dropped by ΔLRU-EDF while ℓ is ineligible. All jobs that are not ineligible are *eligible*.

Let OFF denote an optimal offline algorithm. Let m denote the number of resources given to OFF, where $n = 8m$.We use $Cost_{OFF}(\sigma)$ and $Cost_{\Delta LRU\text{-}EDF}(\sigma)$ to denote the total cost incurred by OFF and ΔLRU-EDF on σ, respectively. We use $ReconfigCost_{OFF}(\sigma)$ and $ReconfigCost_{\Delta LRU\text{-}EDF}(\sigma)$ to denote the reconfiguration cost incurred by OFF and ΔLRU-EDF on σ, respectively. We use $DropCost_{OFF}(\sigma)$ and $DropCost_{\Delta LRU\text{-}EDF}(\sigma)$ to denote the drop cost incurred by OFF and ΔLRU-EDF on σ, respectively. We use $IneligibleDropCost_{\Delta LRU\text{-}EDF}(\sigma)$ to denote the drop cost incurred by ΔLRU-EDF on ineligible jobs in σ. We use $EligibleDropCost_{\Delta LRU\text{-}EDF}(\sigma)$ to denote the drop cost incurred by ΔLRU-EDF on eligible jobs in σ.

We define epochs as follows. For any color ℓ, an *epoch* of ℓ ends the moment ℓ becomes ineligible. A new epoch of ℓ starts when the previous epoch ends. Note that the last epoch of any color can end prematurely. For any input sequence σ, we use $numEpochs(\sigma)$ to denote the total number of epochs (including incomplete epochs) associated with σ. For any color ℓ, we number the epochs of color ℓ from zero. We use $epoch(\ell, j)$ to denote epoch j of color ℓ.

Lemma 3.1 *For any input sequence σ such that for each color ℓ, there are less than Δ color ℓ jobs in σ,*

$$Cost_{\Delta LRU\text{-}EDF}(\sigma) \leq Cost_{OFF}(\sigma).$$

Proof. Consider an arbitrary color ℓ. Since there are less than Δ color ℓ jobs in σ, ℓ never becomes eligible. Hence, ΔLRU-EDF never caches ℓ and drops all color ℓ jobs. If OFF caches ℓ at some point, OFF incurs a reconfiguration cost of Δ. Otherwise, OFF drops all color ℓ job. In either case, the cost incurred by OFF on color ℓ jobs is at least that incurred by ΔLRU-EDF on color ℓ jobs. Summing up over all ℓ's, the lemma follows. ∎

Lemma 3.2 *For any input sequence σ, $EligibleDropCost_{\Delta LRU\text{-}EDF}(\sigma) \leq DropCost_{OFF}(\sigma)$.*

Proof. See Section 3.3. ∎

Lemma 3.3 *For any input sequence σ, $ReconfigCost_{\Delta LRU\text{-}EDF}(\sigma) \leq 4 \cdot numEpochs(\sigma) \cdot \Delta$.*

Proof. We give each epoch 4Δ units of credit: 2Δ units of "first-time" credit and 2Δ units of "end-of-epoch" credit. It is sufficient to show that the total reconfiguration cost incurred by ΔLRU-EDF can be paid for by the credit associated with the epochs.

Consider any color ℓ and any nonnegative integer j. We use the 2Δ units of "first-time" credit associated with $epoch(\ell, j)$ to pay for the reconfiguration cost incurred by the first time ℓ is brought into the cache in $epoch(\ell, j)$ (recall that each time a color is cached, it is cached in two locations). In the following, we show that the reconfiguration cost incurred by each of the subsequent times ℓ is brought into the cache in $epoch(\ell, j)$ can also be paid for.

Until any subsequent time ℓ is brought into the cache in $epoch(\ell, j)$, the deadline of ℓ does not increase since the previous time ℓ is evicted, otherwise ℓ becomes ineligible and $epoch(\ell, j)$ ends. Because the timestamp of ℓ only increases when the deadline of ℓ is reached, the timestamp does not increase since the previous time ℓ is evicted, either. Hence, when ℓ is brought into the cache subsequently in $epoch(\ell, j)$, an idle color ℓ' is evicted. The color ℓ' remains idle until its associated deadline is reached and becomes ineligible; during this period (since the time ℓ' is evicted by ℓ till the time ℓ' becomes ineligible), since ℓ' is idle and its timestamp does not improve, ℓ' remains outside the cache. Therefore, we can associate any subsequent reconfiguration of ℓ in $epoch(\ell, j)$ with the end of an epoch k of a color ℓ', that is, we can use the 2Δ units of "end-of-epoch" credit associated with $epoch(\ell', k)$ to pay for a subsequent reconfiguration of ℓ (in two locations) in $epoch(\ell, j)$.

Summing up over all j's and ℓ's, the lemma follows. ∎

Lemma 3.4 *For any input sequence σ, $IneligibleDropCost_{\Delta\text{LRU-EDF}}(\sigma) \leq numEpochs(\sigma) \cdot \Delta$.*

Proof. Consider any color ℓ and any j. By definition of an epoch, in $epoch(\ell, j)$, ℓ starts off ineligible, becomes eligible, and becomes ineligible again, at which point $epoch(\ell, j)$ ends.

By ΔLRU-EDF, when ℓ becomes ineligible, $\ell.cnt$ is zero. Let k be the round in $epoch(\ell, j)$ during which ℓ becomes eligible. By ΔLRU-EDF, $\ell.cnt$ reaches Δ the first time in $epoch(\ell, j)$ in round k. By the way $\ell.cnt$ is updated, the number of jobs associated with ℓ that arrive in $epoch(\ell, j)$ and before round k is at most Δ. Hence, the ineligible drop cost incurred by ΔLRU-EDF on ℓ in $epoch(\ell, j)$ is at most Δ. Summing up over j's and ℓ's, the lemma follows. ■

Lemma 3.5 *For any input sequence σ such that for each color ℓ, there are at least Δ color ℓ jobs in σ,*

$$Cost_{OFF}(\sigma) = \Omega(numEpochs(\sigma) \cdot \Delta).$$

Proof. See Section 3.4. ■

Theorem 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 be any input sequence for rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$. We break σ into two subsequences α and β, where α consists of jobs of colors with less than Δ jobs in σ, and β consists of the remaining jobs. Let S be the schedule generated by ΔLRU-EDF on σ. Let S' (resp., S'') be the schedule on α (resp., β) obtained by removing jobs in β (resp., α) from S. It is not hard to see that the cost incurred by S' (resp., S'') on α (resp., β) is at most that incurred by S on σ. Let T be the schedule generated by OFF on σ. Let T' (resp., T'') be the schedule on α (resp., β) obtained by removing jobs in β (resp., α) from T. It is not hard to see that the cost incurred by T' (resp., T'') on α (resp., β) is at most that incurred by T on σ.

By Lemma 3.1, the cost incurred by S' on α is at most that incurred by T' on α. By Lemmas 3.2, 3.3, 3.4, and 3.5, the cost incurred by S'' on β is at most a constant factor times that incurred by T'' on β. Hence, the cost incurred by S on σ is at most that incurred by T on σ. The theorem then follows from the fact that $n = 8m$. ■

3.3 Proof of Lemma 3.2

Lemma 3.6 *For any input sequence σ and any subsequence α of σ, $DropCost_{OFF}(\alpha) \leq DropCost_{OFF}(\sigma)$.*

Proof. Since any schedule by OFF on σ implies a schedule by OFF on α of smaller or same drop cost, the lemma follows. ■

We define an algorithm A to be a *double-speed algorithm* if the reconfiguration and execution phases are repeated in each round in A. For any input, a double-speed algorithm produces a *double-speed schedule*. On the other hand, in an *uni-speed algorithm*, the reconfiguration and execution phases are performed only once in each round. For any input, a uni-speed algorithm produces a uni-speed schedule. In this paper, unless otherwise stated, all algorithms and schedules are uni-speed.

In this section, we specify the following schemes of ranking eligible colors and pending jobs, which are invoked by the algorithms that use such schemes in each reconfiguration phase. We rank eligible colors in the same way as used in EDF, that is, rank first on idleness, where nonidle colors comes first, and then in ascending order of deadlines, breaking ties by increasing delay bounds, and then by a consistent order of colors. Note that ΔLRU-EDF uses the same ranking scheme for non-LRU colors that are eligible. We rank pending jobs in increasing order of deadlines, breaking ties by increasing delay bounds, and then by a consistent order of colors. In this paper, we use the same consistent order of colors in all algorithms that use such ranking schemes.

Algorithm Par-EDF is defined as follows. We give m resources to Par-EDF. In each execution phase, we execute up to m pending jobs with the best ranks.

Algorithm Seq-EDF is defined the same as EDF except that Seq-EDF is given m resources and uses all the cache capacity to cache distinct colors, i.e., we do not use half the cache capacity for replication. We use DS-Seq-EDF to denote double-speed Seq-EDF.

For any input sequence σ, we use $DropCost_{\text{Par-EDF}}(\sigma)$ and $DropCost_{\text{DS-Seq-EDF}}(\sigma)$ to denote the drop cost incurred by Par-EDF and DS-Seq-EDF on σ, respectively.

Lemma 3.7 *For any input sequence σ, $DropCost_{\text{Par-EDF}}(\sigma) \leq DropCost_{\text{OFF}}(\sigma)$.*

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

We define a *mini-round* to be an iteration of the reconfiguration and execution phases in a round. We number the mini-rounds from zero. By definition, there are two (resp., one) mini-rounds in a round in any double-speed (resp., uni-speed) algorithm or schedule. We define a *slot* to be a mini-round j on a resource k, identified by slot (k, j). A slot (k, j) is *occupied* if a job is scheduled in mini-round j and on resource k. A slot that is not occupied is *free*. we define a *column* to be the set of slots in the same mini-round. A column is *full* if all slots in the column are occupied, and *nonfull* otherwise. Columns are ordered in increasing order of mini-round indices.

For any delay bound p, we define *blocks* of delay bound p as follows. For any nonnegative integer i, block i of delay bound p, denoted by $block(p, i)$, is the p rounds starting from round $i \cdot p$.

An input σ is defined to be *nice* if Par-EDF does not incur any drops on σ.

Lemma 3.8 *For any nice input sequence σ, DS-Seq-EDF does not incur any drops on σ.*

Proof. The proof proceeds in two steps as follows. First, we construct a double-speed schedule T that executes all jobs in σ. Second, we show that T is a schedule generated by DS-Seq-EDF.

In the first step, we schedule jobs in increasing order of delay bounds. For a certain delay bound p, we schedule jobs with delay bound p block by block. For a certain block of p, we schedule jobs with delay bound p in the consistent order of colors mentioned above. We now describe the scheduling process for any delay bound p, any block i of p, and any color ℓ with delay bound p. Let X be the set of color ℓ jobs that arrive in $block(p, i)$. First, we pick the first $|X|$ nonfull columns. Second, in each of the columns picked in the first step, we pick an arbitrary free slot. Third, we schedule X in the $|X|$ slots picked in the second step.

We need to show that there are at least $|X|$ nonfull columns while we schedule X. By definition of rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, $|X| \leq p$, hence it is sufficient to show that at least p nonfull columns while we schedule X, which we prove as follows. Let S be the schedule generated by Par-EDF on σ. Since σ is a nice schedule, all jobs that arrive in $block(p, i)$ are executed by S. Since T is a double-speed schedule, the number of slots in $block(p, i)$ in T is twice that in S. Hence, the number of slots in $block(p, i)$ in T is at least twice the number of jobs that arrive in $block(p, i)$. Hence at least half of the columns, that is, at least p columns, are nonfull while we schedule X.

In the second step, we show that T is a schedule generated by DS-Seq-EDF as follows. Since all delay bounds are powers of 2, the increasing order of delay bounds agrees with the increasing order of deadlines. Hence, the ranking of nonidle eligible colors agrees with the increasing order of delay bounds. By the construction of T, in each mini-round, a job of color ℓ is not scheduled until one job of each nonidle eligible color is scheduled that has a larger delay bound or has same delay bound and precedes ℓ in the consistent order of colors. Hence, T is a schedule generated by DS-Seq-EDF. ∎

Lemma 3.9 *For any input sequence σ and any subsequence α of σ, if DS-Seq-EDF executes j jobs when operated on α, then DS-Seq-EDF executes at least j jobs when operated on σ.*

Proof. Let $\beta = \sigma \setminus \alpha$. We sort jobs in β in increasing order of arrival time, breaking ties arbitrarily. We define $\gamma_0 = \alpha$. For $0 \leq i < |\beta|$, we define β_i to be job i in β and $\gamma_{i+1} = \gamma_i \cup \{\beta_i\}$. By definition, $\sigma = \gamma_{|\beta|}$.

In the following, we prove the lemma by showing that, for any i such that $0 \leq i < |\beta|$, $|X_i| \leq |X_{i+1}|$, where X_i is the set of jobs executed by DS-Seq-EDF when operated on γ_i. If $\beta_i \notin X_{i+1}$, $X_{i+1} = X_i$. Otherwise, $|X_i \setminus X_{i+1}| \leq 1$. In either case, $|X_i| \leq |X_{i+1}|$. This completes the proof and the lemma follows. ∎

Corollary 3.1 *For any input sequence σ, $DropCost_{\text{DS-Seq-EDF}}(\sigma) \leq DropCost_{\text{Par-EDF}}(\sigma)$.*

Proof. If σ is nice, the corollary follows immediately from Lemma 3.8. Otherwise, we break σ into two subsequences α and β, where α consists of the jobs executed by Par-EDF on σ, and β consists of the remaining jobs, that is, the jobs dropped by Par-EDF on σ. By Lemma 3.8, DS-Seq-EDF does not incur any drops on α. By Lemma 3.9, the number of jobs executed by DS-Seq-EDF on σ is at least the number of jobs executed by DS-Seq-EDF on α. Hence the corollary follows. ∎

Lemma 3.10 *Consider any input sequence σ. Let α be the subsequence of σ that consists of eligible jobs in σ. Then $EligibleDropCost_{\Delta\text{LRU-EDF}}(\sigma) \leq DropCost_{\text{DS-Seq-EDF}}(\alpha)$.*

Proof. Consider ΔLRU-EDF and DS-Seq-EDF proceed concurrently. Let X_i (resp., Y_i) be the set of pending eligible jobs in ΔLRU-EDF (resp., DS-Seq-EDF) at the beginning of round i. We show the lemma by proving that for any i, $X_i \subseteq Y_i$.

The proof is obtained by induction. For the base case, $i = 0$. It is obvious that $X_0 = Y_0 = \emptyset$. The induction step is as follows. Suppose $X_i \subseteq Y_i$, we show in the following that $X_{i+1} \subseteq Y_{i+1}$. Let X'_i and Y'_i be the set of pending eligible jobs in ΔLRU-EDF and DS-Seq-EDF at the end of the arrival phase in round i. In the arrival phase of round $i + 1$, the number of newly arrived eligible jobs become pending in both algorithms. This observation, together with the induction hypothesis that $X_i \subseteq Y_i$, indicates that $X'_{i+1} \subseteq Y'_{i+1}$. Let color ℓ be any color that is ever configured by DS-Seq-EDF in round $i + 1$. By definition of DS-Seq-EDF, color ℓ is among the $2m$ nonidle eligible colors with the best ranks, and in round $i + 1$, DS-Seq-EDF executes up to 2 jobs of color ℓ. Since $X'_{i+1} \subseteq Y'_{i+1}$, 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 $n = 4m$, i.e., $2m = \frac{n}{4}$, by 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. In round $i + 1$, DS-Seq-EDF configures up to $2m$ distinct colors, ΔLRU-EDF configures $2m$ distinct nonidle colors if there are that many, and all nonidle colors otherwise. Therefore, $X_{i+1} \subseteq Y_{i+1}$. ∎

Proof of Lemma 3.2. Consider any input sequence σ. Let α be the subsequence of σ that consists of the eligible jobs in σ. By Lemma 3.6, $DropCost_{\text{OFF}}(\alpha) \leq DropCost_{\text{OFF}}(\sigma)$. By Lemmas 3.10 and 3.7 and Corollary 3.1, $DropCost_{\Delta\text{LRU-EDF}}(\sigma) \leq DropCost_{\text{OFF}}(\alpha)$. Hence, the lemma follows. ∎

3.4 Proof of Lemma 3.5

We define super-epochs as follows. A *super-epoch* ends the moment that at least $2m$ colors increase their timestamps since the start of the current super-epoch. A new super-epoch starts when the previous super-epoch ends. Note that the last super-epoch can end prematurely. For convenience, we number the super-epochs from zero.

We define a color ℓ to be an *i-active color* if the timestamp of ℓ is updated in super-epoch i. For any i-active color ℓ, any epoch of ℓ that overlaps with super-epoch i is defined to be an *i-active epoch*. We define an epoch to be *special* if it is not i-active for any complete super-epoch i. An epoch that is not special is *nonspecial*.

We attribute jobs to counter wrapping events as follows. For any color ℓ and any round k that is an integral multiple of D_ℓ, let X be the set of color ℓ jobs that arrive in round k and j be the value of $\ell.cnt$ at the beginning of round k. If $|X| < \Delta - j$, there is no counter wrapping event of color ℓ in round k; we attribute all jobs in X to the next counter wrapping event of color ℓ. Otherwise, there is a counter wrapping event of color ℓ in round k; we attribute any $\Delta - j$ jobs in X to the counter wrapping event of color ℓ in round k, and the rest jobs in X to the next counter wrapping event of color ℓ.

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

Lemma 3.11 *The number of jobs attributed to each counter wrapping event is at least Δ.*

We define a timestamp update event of color ℓ to be the event that the timestamp of ℓ is updated. We assign credit to timestamp update events as follows: (1) if color ℓ is i-active and there is a reconfiguration from or to color ℓ in super-epoch i incurred by OFF, we give 6Δ units of credit to the first timestamp update event of color ℓ in super-epoch i; (2) for each reconfiguration from or to a color ℓ incurred by OFF, we give 6Δ units of credit to each of the next two timestamp update events of color ℓ; (3) for any color ℓ job x that is dropped by OFF, we give 6 units of credit to the first timestamp update event of color ℓ subsequent to the counter wrapping event which x is attributed to, if such events exists.

The following lemma follows from the way we assign credit.

Lemma 3.12 *The total credit associated with timestamp update events over all colors is $O(Cost_{\text{OFF}}(\sigma))$.*

Lemma 3.13 *For any i-active color ℓ, either ℓ is cached throughout super-epoch i, or there are at least 6Δ units of credit associated with the first timestamp update event of ℓ in super-epoch i.*

Proof. Let k be the index of the round in which the first timestamp update event of ℓ in super-epoch i occurs. Let r be the index of the round from which super-epoch i starts. If at least two counter wrapping events of color ℓ occur before round k, we define j to be the index of the round in which the second to last counter wrapping event of ℓ before round k occurs. Otherwise, we define j to be 0.

We first show that $j \leq r$, which we use later in the proof of the lemma. If less than two counter wrapping events of ℓ occur before round k, $j = 0$ and the claim holds. Otherwise, we show the claim as follows. Let j' be the index of the round in which the last counter wrapping event of ℓ before round k occurs. By the definitions of counter wrapping events, timestamps, and timestamp update events, $j < j' < k$, and the timestamp of ℓ is updated once between round j and round j' (including round j'). Since the first timestamp update event of ℓ in super-epoch i occurs in round k, $j \leq r$.

Let V be the time interval between round j and round r, which is well-defined by the claim shown above. We now prove the lemma as follows. If OFF evicts ℓ from the cache or brings ℓ into the cache in super-epoch i, by credit assignment rule (1), the first timestamp update event of ℓ in super-epoch i gets 6Δ units of credit. If OFF keeps ℓ out of the cache throughout super-epoch i, we consider the following two cases.

- Algorithm OFF evicts ℓ out of the cache or brings ℓ into the cache in V. It is not hard to see that there is at most one timestamp update event of ℓ in V. Hence, the first timestamp update event of ℓ in super-epoch i is either the first or the second timestamp update events subsequent to any reconfiguration in V. By credit assignment rule (2), the first timestamp update event of ℓ in super-epoch i gets 6Δ units of credit.
- Algorithm OFF keeps ℓ out of the cache in V. Then OFF keeps ℓ out of the cache since round j till round k. In this case, all jobs contributed to the last counter wrapping event of ℓ before round k are dropped by OFF. By Lemma 3.11 and credit assignment rule (3), the first timestamp update event of ℓ in super-epoch i gets at least 6Δ units of credit.

Hence, either ℓ is either cached throughout super-epoch i, or the first timestamp update event of ℓ in super-epoch i gets at least 6Δ units of credit. ■

Lemma 3.14 *For any color ℓ and and integer j, the timestamp of ℓ is updated in $epoch(\ell, j)$, and at the end of $epoch(\ell, j)$, the timestamp is at least the start of $epoch(\ell, j)$.*

Proof. In $epoch(\ell, j)$, ℓ starts off ineligible, becomes eligible, and becomes ineligible again, at which point $epoch(\ell, j)$ ends. At the time ℓ becomes eligible, the counter of ℓ is wrapped around. At the time ℓ becomes ineligible again, the current deadline of ℓ is reached. By definition of the timestamp, the lemma follows. ■

Lemma 3.15 *For any super-epoch i and any color ℓ, once ℓ has two complete epochs in super-epoch i, super-epoch i ends.*

Proof. By definition of the timestamp, the value of the timestamp of any color is smaller than current time. Hence, at the beginning of super-epoch i, the timestamp of any color is smaller than the start of super-epoch i. By definition of a super-epoch, at most $2m$ colors increase their timestamps during super-epoch i (excluding the end of super-epoch i). Hence, until the end of super-epoch i, at most $2m$ colors have timestamps with value at least the start of super-epoch i.

Once ℓ has a complete epoch super-epoch i, by Lemma 3.14, the timestamp of ℓ is at least the start of super-epoch i, which indicates ℓ is among the $2m = \frac{n}{4}$ colors with the most recent timestamps. In the second complete epoch ℓ in super-epoch i, when ℓ becomes eligible, ℓ is brought into the cache and kept in until the super-epoch i ends.

By definition of epochs, at the end of the second complete epoch of ℓ in super-epoch i, ℓ becomes ineligible. Because a color can only become ineligible when it is out of the cache, ℓ is out of the cache when the second epoch ends, which means super-epoch i has ended. Hence the lemma follows. ■

The following corollary immediately from Lemma 3.15.

Corollary 3.2 *For any color ℓ and any nonnegative integer i, there are at most three epochs of color ℓ that overlap with super-epoch i.*

Lemma 3.16 *For each color ℓ, there are at most three special epochs.*

Proof. By Lemma 3.14 and the definition of an i-active epoch, a complete epoch contained in any super-epoch i is i-active. Hence, a special epoch is either incomplete, or overlaps with the incomplete super-epoch. The lemma then follows from Corollary 3.2 and the fact that there is only one incomplete epoch and one incomplete super-epoch. ■

Corollary 3.3 *For any input sequence σ such that for each color ℓ, there are at least Δ color ℓ jobs in σ, $Cost_{OFF}(\sigma)$ is at least 3Δ times the number of special epochs.*

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. The corollary then follows from Lemma 3.16. ∎

Lemma 3.17 *The total credit associated with the timestamp update events is at least Δ times the number of nonspecial epochs.*

Proof. Let $X = \{j \mid \text{super-epoch } j \text{ is complete}\}$. 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 of which the first timestamp event in super-epoch i is assigned at least 6Δ units of credit. Let k_i'' be the number of i-active colors that are cached throughout super-epoch i. By Lemma 3.13, $k_i \leq k_i' + k_i''$. By 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,

$$k_i \leq 2k_i'. \tag{1}$$

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

$$\begin{aligned}
\text{number of nonspecial epochs} &\leq \sum_{i \in X} q_i \\
&= \sum_{i \in X} \sum_{\ell} q_{i,\ell} \\
&\leq 3 \sum_{i \in X} k_i \\
&\leq 6 \sum_{i \in X} k_i'.
\end{aligned}$$

(The first inequality follows from the definition of i-active epochs and nonspecial epochs. The second equality uses the definitions of i-active colors and i-active epochs. The third inequality follows from the definitions of i-active colors, i-active epochs and Corollary 3.2. The last inequality uses Equation (1).) Obviously, the total credit is at least $6\Delta \sum_{i \in X} k_i'$, hence the lemma follows. ∎

Lemma 3.5 immediately follows from Corollary 3.3, Lemmas 3.12 and 3.17.

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 variant is characterized by a fixed configuration cost Δ, a unit drop cost, a per-color delay bound D_ℓ, and batched arrivals (jobs with delay bound D_ℓ arrive at integral multiples of D_ℓ).

The solution to this variant uses a reduction to rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, which is solved in Section 3.

4.1 Algorithm Distribute

Algorithm Distribute proceeds in three steps. In the first step, given an arbitrary instance I of $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, where D_ℓ is a power of 2, we construct an instance I' of rate-limited $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$ as follows. Each color associated with I' is characterized by a color ℓ associated with I and a nonnegative integer j, denoted by (ℓ, j). Let σ be the input sequence associated with I. For any nonnegative integer i, let σ_i be request i of σ. For any color ℓ, we rank color ℓ jobs in σ_i in an arbitrary order. For any color ℓ and any color ℓ job x in σ_i, we construct a job y with the same characterization except the color of y is (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. The input sequence σ' that associates with I' is the the concatenation of σ_i' in increasing order of i.

In the second step, we use algorithm ΔLRU-EDF to obtain a schedule S' for I'.

In the third step, we construct a schedule S for I from S' as follows. Whenever S' configures color (ℓ, j), S configures color ℓ. Whenever S' executes a job of color (ℓ, j), S executes a job of color ℓ.

Note that Distribute is an online algorithm.

4.2 Analysis

Lemma 4.1 *If there exists an offline schedule T for I, then there exists an offline schedule T' for I' that is resource competitive with T.*

Proof. See Section 4.3. ∎

Lemma 4.2 *The cost incurred by S is at most that incurred by S'.*

Proof. Since S replaces color (ℓ, j) with color ℓ, the reconfiguration cost incurred by S is at most that incurred by S'. Since the number of ℓ jobs executed by S equals the number (ℓ, j) jobs executed by S', hence the drop cost incurred by S equals that incurred by S'. Hence, the lemma follows. ∎

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

Proof. Suppose there exists an offline schedule T for I. By Lemma 4.1, there exists an offline schedule T' for I' that is resource competitive with T. By Theorem 1, the schedule S', that is, the schedule given by algorithm ΔLRU-EDF for I', is resource competitive with T'. By Lemma 4.2, the cost incurred by S, that is, the schedule obtained by algorithm Distribute for I, is at most that incurred by S'. Hence, S is resource competitive with T and the theorem follows. ∎

4.3 Proof of Lemma 4.1

In this section, we use the definitions of blocks and slots, which are defined in Section 3.3. We sort slots in ascending order of resource indices and then in ascending order of mini-round indices, where mini-rounds are defined in Section 3.3.

For any schedule S, any delay bound p and any nonnegative integer i, we define a resource k to be (S, p, i)-*monochromatic* if resource k is configured with one color throughout $block(p, i)$ in S, and (S, p, i)-*multichromatic* otherwise. An (S, p, i)-monochromatic resource k is defined to be (S, p, i, ℓ)-*monochromatic* if resource k is configured with color ℓ throughout $block(p, i)$ in S.

In the following, we introduce an algorithm Aggregate that takes an arbitrary schedule T as input, and generates a schedule T' with three times the resources of T. For convenience, with each resource k in T, we associate resources $3k$, $3k + 1$, and $3k + 2$ in T', referred to as resource $(k, 0)$, $(k, 1)$, and $(k, 2)$, respectively. We use $X_{p,i}$ and $Y_{p,i}$ to denote the the set of resources that are (T, p, i)-monochromatic and (T, p, i)-multichromatic, respectively. We define $X'_{p,i}$ to be

$$\{\text{resource } (k, 0),\ (k, 1),\ \text{and } (k, 2) \mid \text{resource } k \in X_{p,i}\}$$

and $Y'_{p,i}$ to be

$$\{\text{resource } (k, 0),\ (k, 1),\ \text{and } (k, 2) \mid \text{resource } k\ \in Y_{p,i}\}.$$

We define $M_{p,i,\ell}$ to be

$$\{\text{resource } (k, 0) \mid \text{resource } k \text{ is } (T, p, i, \ell)\text{-monochromatic}\}.$$

For any resource $(k, 0)$ in $M_{p,i,\ell}$, we define its T-*level* in $block(p, i)$ to be the largest delay bound q such that resource k is (T, q, j)-monochromatic, where $block(q, j)$ encloses $block(p, i)$. Resources in $M_{p,i,\ell}$ are ranked in descending order of T-levels in $block(p, i)$.

To construct T', Aggregate starts with an empty schedule and schedules all jobs executed by T by proceeding in ascending order of delay bounds. For a certain delay bound p, Aggregate proceeds block by block in increasing order block indices. For a certain block of p, Aggregate proceeds in an arbitrary order of colors with delay bound p. Now we describe Aggregate for any delay bound p, any block i of p, and any color ℓ with delay bound p.

First, we label the resources in $M_{p,i,\ell}$ from 0 to $|M_{p,i,\ell}| - 1$ as follows. If $i = 0$, we label resources in $M_{p,i,\ell}$ from 0 to $|M_{p,i,\ell}| - 1$ arbitrarily. Otherwise, for any resource k such that resource $(k, 0)$ is in both $M_{p,i,\ell}$ and $M_{p,i-1,\ell}$, we let resource $(k, 0)$ inherit its label in block $(p, i - 1)$; we then give the remaining labels in $[0, |M_{p,i,\ell}|)$ to the remaining resources in $M_{p,i,\ell}$, one label per resource.

Second, we partition the set of color ℓ jobs executed by T in $block(p, i)$ into groups of size p (one of the groups can have size less than p).

Third, we assign groups of color ℓ to the resources in $M_{p,i,\ell}$ in descending order of group size and in descending order of resource ranks, one group per resource.

Fourth, we determine the schedule of resources in $M_{p,i,\ell}$ in $block(p,i)$ as follows. For any resource $(k,0)$ in $M_{p,i,\ell}$ to which we assign a group U, we execute $|U|$ color (ℓ,j) jobs continuously on resource $(k,0)$ in $block(p,i)$, and then mark all slots on resource $(k,0)$ in $block(p,i)$ as occupied, where j is the label we give to resource $(k,0)$ for $block(p,i)$ in the first step.

Fifth, we set $q = |M_{p,i,\ell}|$. While there is at least one group not assigned yet, we perform the following steps.

1. We pick an arbitrary k such that resource $(k,0)$, $(k,1)$ and $(k,2)$ are in $Y'_{p,i}$ and there are at least p free slots in $block(p,i)$ on them (we will show such k exists in Lemma 4.4).

2. Let U be the group not assigned with the largest size (breaking ties arbitrarily). We assign U to resource $(k,0)$, $(k,1)$ and $(k,2)$, execute $|U|$ color (ℓ,q) jobs in the first free $|U|$ slots in $block(p,i)$ on resource $(k,0)$, $(k,1)$ and $(k,2)$, and increment q.

Lemma 4.3 *The schedule T' is a schedule for I'.*

Proof. By the construction of I' and T', it is not hard to see that the jobs executed by T' is a subset of the jobs in σ', the input associated with I'. Hence the lemma follows. ∎

Lemma 4.4 *For any delay bound p and any nonnegative integer i, while algorithm Aggregate works on the schedule of jobs with delay bound p in $block(p,i)$, there exists k such that resource $(k,0)$, $(k,1)$ and $(k,2)$ are in $Y'_{p,i}$ and there are at least p free slots in $block(p,i)$ on these three resources.*

Proof. We fix our attention on the process of scheduling jobs with delay bound p in $block(p,i)$.

First, we show that all jobs scheduled to $block(p,i)$ in or before this process are executed in $block(p,i)$ in T. To see that, we observe that (1) for any delay bound q and any nonnegative integer j, all jobs of delay bound q scheduled to $block(q,j)$ are executed in $block(q,j)$ in T; (2) in or before this process, only jobs of delay bound at most p are scheduled. The claim follows from these two observations, and the fact that each delay bound is a power of 2.

Second, we show that the number of jobs we schedule on resources in $X'_{p,i}$ in $block(p,i)$ is at least that executed on resources in $X_{p,i}$ in $block(p,i)$ in T. It is obvious that the claim holds for jobs with delay bound p. It remains to show the claim for jobs with delay bound less than p. Let q be any delay bounds less than p. Let j be any nonnegative integer such that $block(q,j) \subset block(p,i)$. Let color ℓ be any color with delay bound q. Let r be the number of color ℓ jobs arrive in $block(q,j)$. We define $X_{q,j,\ell}$ to be the (T,q,j,ℓ)-monochromatic resource. By the way we schedule color ℓ jobs, we fill resources in $M_{q,j,\ell}$ with color ℓ jobs in descending order of T-levels in $block(q,j)$. By definitions of T-levels, the T-level of any resource in $M_{q,j,\ell} \cap X'_{p,i}$ in $block(q,j)$ is greater than that of any resource in $M_{q,j,\ell} \cap Y'_{p,i}$ in $block(q,j)$. Hence, the number of color ℓ scheduled on resources in $M_{q,j,\ell} \cap X'_{p,i}$ in $block(q,j)$ is $\min\left(q \cdot \left|M_{q,j,\ell} \cap X'_{p,i}\right|, r\right)$. It is easy to see that the number of color ℓ scheduled on resources in $X_{q,j,\ell} \cap X_{p,i}$ in $block(q,j)$ is at most $\min\left(q \cdot \left|X_{q,j,\ell} \cap X_{p,i}\right|, r\right)$. By definition of $X_{p,i}$, $X'_{p,i}$, $M_{q,j,\ell}$, and $X_{q,j,\ell}$, $|X_{q,j,\ell} \cap X_{p,i}| = \left|M_{q,j,\ell} \cap X'_{p,i}\right|$. Hence, the number of color ℓ jobs scheduled on resources in $M_{q,j,\ell} \cap X'_{p,i}$ in $block(q,j)$ is at least that executed on resources in $X_{q,j,\ell} \cap X_{p,i}$ in $block(q,j)$ in T. By definition of $X_{p,i}$, $X_{q,j,\ell}$, and the fact that color ℓ is associated with delay bound q, no color ℓ jobs are executed on resources in $X_{p,i} \setminus X_{q,j,\ell}$ in $block(q,j)$. Hence, the claim holds for color ℓ jobs in $block(q,j)$. Summing up over all ℓ's, j's and q's, the claim follows.

From the above two steps, we conclude that the number of jobs scheduled in $Y'_{p,i}$ in $block(p,i)$ is at most that executed in $Y_{p,i}$ in $block(p,i)$ in T, that is, the number of jobs scheduled in $Y'_{p,i}$ in $block(p,i)$ is at most $p \cdot |Y_{p,i}|$. Since the total number of slots that are marked as occupied in $Y'_{p,i}$ is at least the $p \cdot \left|Y'_{p,i}\right|$, and $\left|Y'_{p,i}\right| = 3\,|Y_{p,i}|$, we obtain that at least $\frac{1}{3}$ of the slots in $Y'_{p,i}$ are free. Hence the lemma follows. ∎

Lemma 4.5 *The drop cost incurred by T' is the same as that incurred by T.*

Proof. It is sufficient to show that, for any delay bound p and nonnegative integer i, the jobs of delay bound p executed by T in $block(p,i)$ are executed by T'. By Aggregate, we intend to schedule all jobs of delay bound p executed by T in $block(p,i)$, it is sufficient to show that whenever we schedule a group of jobs of delay bound p, there are enough slots in the target resources. It is not hard to see that the jobs of delay bound p scheduled to $X'_{p,i}$ can find enough slots in the target resources. By Lemma 4.4, we conclude that the jobs of delay bound p scheduled to $Y'_{p,i}$ can find enough slots in the target resources, too. Hence, the lemma follows. ∎

Lemma 4.6 *The reconfiguration cost incurred by T' is at most a constant factor of that incurred by T.*

Proof. We define a reconfiguration in T' to be a *special reconfiguration* if the reconfiguration is made on the boundary of $block(p, i)$ on resources in $M_{p,i,\ell}$, for any p, i and ℓ. For any group U of jobs with delay bound p that are executed in $block(p, i)$, we define U to be a (p, i)-multichromatic group if U is assigned to resources in $Y'_{p,i}$.

We first bound the cost incurred by special reconfigurations. Fix an arbitrary delay bound p and a nonnegative integer i. We consider special reconfigurations on the boundary between $block(p, i)$ and $block(p, i+1)$. It is not hard to verify that if resource k is (T, p, i, ℓ)-monochromatic, then resource $(k, 0)$ is $(T', p, i, (\ell, j))$-monochromatic, for some nonnegative integer j. Hence, it is sufficient to bound the reconfiguration cost incurred by T due to the relabeling of resources in $\cup_\ell Y_{p,i,\ell}$, where $Y_{p,i,\ell}$ is the set of resources that are in both $M_{p,i,\ell}$ and $M_{p,i+1,\ell}$. Fix any color ℓ. Let $r_{p,i,\ell}$ be the number of (T, p, i, ℓ) resources. Let $q_{p,i,\ell}$ be the number of (T, p, i, ℓ) resources with labels at least $r_{p,i+1,\ell}$ in $block(p, i)$. It is not hard to verify that the number of resources in $Y_{p,i,\ell}$ that change labels from $block(p, i)$ to $block(p, i+1)$ is at most $q_{p,i,\ell}$, which is in turn at most $\max(0, r_{p,i,\ell} - r_{p,i+1,\ell})$. Summing up over all ℓ's, we obtain that the number of resources that change labels from $block(p, i)$ to $block(p, i + 1)$ is at most the number of reconfigurations on the boundary between $block(p, i)$ and $block(p, i + 1)$ in T. Summing up over all p's and i's, the cost incurred by special reconfigurations is at most the reconfiguration cost incurred by T.

We then bound the cost incurred by nonspecial reconfigurations in the following three steps. First, we associate 6Δ units of credit with each reconfiguration in T. Second, we show that we can spread the credit so that each (p, i)-multichromatic group gets 6Δ units of credit, for any delay bound p and any nonnegative integer i, as follows. By the way we form groups and the fact that we can execute at most p jobs on a resource in $block(p, i)$, the number of (T, p, i)-multichromatic resources is at least the number of (p, i)-multichromatic groups. Since there is at least one reconfiguration on each (T, p, i)-multichromatic resource in $block(p, i)$, hence the claim follows. Third, we show that the cost incurred by nonspecial reconfigurations can be bounded by the total credit associated with the multichromatic groups as follows. For each multichromatic group U of jobs with delay bound p that are executed in $block(p, i)$, we use 2Δ units of credit to pay for the reconfigurations at the beginning and end of U in T', and 4Δ to pay for the reconfigurations caused by the wrapping around when the end of $block(p, i)$ is encountered while scheduling U. Hence the cost incurred by nonspecial reconfigurations in T' is within a constant factor of the reconfiguration cost incurred by T.

Hence, the lemma follows. ∎

Lemma 4.1 follows from Lemmas 4.3, 4.5, and 4.6.

5 Our Main Result

In this section, we solve $[\Delta \mid 1 \mid D_\ell \mid 1]$, which is characterized by a fixed configuration cost Δ, a unit drop cost, per-color delay bound D_ℓ, and non batched 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. The special case is solved by a reduction to $[\Delta \mid 1 \mid D_\ell \mid D_\ell]$, which is solved in Section 4. For any color ℓ such that $D_\ell = 1$, jobs of color ℓ are already batched. Hence, throughout this section, we assume $D_\ell > 1$, for any color ℓ. Section 5.1 and Section 5.2 give the algorithm and analysis for the reduction, respectively. In Section 5.3, we comment on how to extend our solution for the special case to arbitrary delay bounds.

5.1 Algorithm VarBatch

For any delay bound p, we define *half-blocks* of delay bound p as follows. For any nonnegative integer i, half-block i of delay bound p, denoted by $halfBlock(p, i)$, is the $\frac{p}{2}$ rounds starting from round $i \cdot \frac{p}{2}$.

Algorithm VarBatch takes an input sequence σ for $[\Delta \mid 1 \mid D_\ell \mid 1]$, where each D_ℓ is a power of 2, and proceeds in the following two steps. First, we construct an input σ' for $[\Delta \mid 1 \mid \frac{D_\ell}{2} \mid \frac{D_\ell}{2}]$ by delaying any job x of delay bound p that arrives in $halfBlock(p, i)$ until $halfBlock(p, i+1)$, and restricting the execution of x to $halfBlock(p, i+1)$. Second, we apply algorithm Distribute on σ' to obtain the final schedule.

Note that algorithm VarBatch is an online algorithm.

5.2 Analysis of VarBatch

Consider any delay bound p and any job x of delay bound p. Let x arrive in $halfBlock(p, i)$. We say the execution of x is *early* if x is executed in $halfBlock(p, i)$, *punctual* if x is executed in $halfBlock(p, i + 1)$, and *late* if x is executed in

$halfBlock(p, i+2)$. We define a schedule S to be *early* (resp., *late*) if all job executions in S are early (resp., late). We define a schedule S to be *punctual* if all job executions in S are punctual.

Lemma 5.1 *For any input sequence σ and any early offline schedule S with reconfiguration cost C and one resource, there exists a punctual schedule S' that executes all jobs executed by S with three resources and incurs a reconfiguration cost of $O(C)$.*

Proof. Given S, we construct S' in the following three steps. First, we identify a set of jobs as *special* jobs as follows. For any delay bound p, any color ℓ with delay bound p and any nonnegative integer i, if color ℓ is configured throughout $halfBlock(p, i)$ and $halfBlock(p, i+1)$ in S, we label all color ℓ jobs executed in $halfBlock(p, i)$ in S as special.

Second, we schedule special jobs on resource 0 as follows. For any color ℓ and special job x of color ℓ executed in round j in S, we execute x in round $j + \frac{D_\ell}{2}$ on resource 0 of S'.

Third, we determine the schedule of nonspecial jobs on resource 1 and 2 in ascending order of delay bounds. For any delay bound p and any nonnegative integer i, we determine the schedule of nonspecial jobs of delay bound p in $halfBlock(p, i)$ in an arbitrary order of colors as follows. For any color ℓ with delay bound p, let X_ℓ be the set of nonspecial jobs of color ℓ executed in $halfBlock(p, i)$ in S. We schedule jobs in X_ℓ in the first free slots on resource 1 and 2 in $halfBlock(p, i+1)$, where slots are defined in Section 3.3.

We need to show the following properties of S': (1) all job executions in S' are punctual; (2) the drop cost incurred by S' is the same as that incurred by S; (3) the reconfiguration cost incurred by S' is at most a constant factor that incurred by S.

By the way we schedule jobs in S', for any delay bound p and nonnegative integer integer i, any job of delay bound p executed in $halfBlock(p, i+1)$ in S' arrives in block $halfBlock(p, i)$. By definition of punctual executions and the fact that S is an early schedule, all job executions in S' are punctual, that is, (1) holds.

To show (2), it is sufficient to show that any job executed by S is executed by S'. It is straightforward that any special job executed by S is executed by S'. In the following, we show that any nonspecial jobs executed by S is executed by S'. It is not hard to verify that, for any delay bound p and any nonnegative integer i, nonspecial job scheduled in $halfBlock(p, i+1)$ in S' are executed in $halfBlock(p, i)$ or $halfBlock(p, i+1)$ in S, hence with two resources, we can execute all nonspecial job scheduled to $halfBlock(p, i+1)$ in S'. Summing up over all p's and i's, we obtain that any nonspecial job executed by S is executed by S'. Therefore, (2) follows.

It is straightforward that the reconfiguration cost incurred on resource 0 of S' is at most the reconfiguration cost incurred by S. Now we show how to bound the reconfiguration cost incurred on resource 1 and 2 of S' with the accounting method in the following three steps. First, we associate 12Δ units of credit with each reconfiguration in S. Second, we show that we can spread the credit such that each group gets at least 4Δ units of credit, where a group is a continuous sequence of nonspecial jobs of the same color ℓ in a half-block of D_ℓ in S, as follows. For each reconfiguration from color ℓ to color ℓ' on a resource k, we give 4Δ units of credit to the group of color ℓ' scheduled on resource k immediately after the reconfiguration in $block(D_{\ell'}, j)$; we give 4Δ units of credit to the group of color ℓ scheduled on resource k immediately before the reconfiguration in $block(D_\ell, i)$; we give 4Δ units of credit to the last group of color ℓ scheduled on resource k in $block(D_\ell, i-1)$; where the reconfiguration incurs in $block(D_\ell, j)$ and in $block(D_{\ell'}, j)$. Third, we show that the total credit associated with the groups can pay for the reconfiguration cost on resource 1 and 2 in S' as follows. For each group U of color ℓ, we use 2Δ units of credit to pay for the beginning and end of U in S', and 2Δ to pay for the reconfigurations caused by the wrapping around when the end of the current half-block of D_ℓ is encountered while scheduling U. Hence, (3) follows.

Therefore, the lemma follows. ∎

The following lemma can be proved with a proof similar to that for Lemma 5.1 and hence omitted here.

Lemma 5.2 *For any input sequence σ and any late offline schedule S with cost C and one resource, there exists a punctual schedule S' that executes all jobs executed by S with three resources and incurs a reconfiguration cost of $O(C)$.*

Lemma 5.3 *For any input σ for $[\Delta \mid 1 \mid D_\ell \mid 1]$ and any offline schedule S for σ, there exists a punctual schedule S' that is resource competitive with S.*

Proof. Suppose S uses m resources. Consider any integer k such that $0 \le k < m$. Let S_k denote the schedule of resource k in S. Let C_k denote the cost incurred by S_k. Let $S_{k,0}$, $S_{k,1}$, and $S_{k,2}$ denote the schedule obtained by retaining only the early, punctual, and late executions in S_k, respectively. Obviously, the reconfiguration cost incurred by each of $S_{k,0}$, $S_{k,1}$, and $S_{k,2}$ is at most C_k.

By Lemma 5.1, there exists a punctual schedule $S'_{k,0}$ that executes all jobs executed by $S_{k,0}$ with three resources and incurs a reconfiguration cost of $O(C_k)$. By Lemma 5.2, there exists a punctual schedule $S'_{k,2}$ that executes all jobs executed by $S_{k,2}$ with three resources and incurs a reconfiguration cost of $O(C_k)$. Hence, all jobs executed by S_k are executed by $S'_{k,0}$, $S_{k,1}$ and $S'_{k,2}$, and the total reconfiguration cost incurred by $S'_{k,0}$, $S_{k,1}$, and $S'_{k,2}$ are $O(C_k)$.

Given $7m$ resources, we construct S' as follows. We use resources $7k$ to $7k+6$ to execute the jobs executed on resource k in S. We use $S'_{k,0}$ in resources from $7k$ to $7k+2$, $S_{k,1}$ in resources $7k+3$, $S'_{k,2}$ in resources from $7k+4$ to $7k+6$. By the above argument, all jobs executed in S_k are executed on resources from $7k$ to $7k+6$ in S', and the total reconfiguration cost incurred by S' on resources from $7k$ to $7k+6$ are $O(C_k)$. Summing up over all k's, the lemma follows. ∎

Theorem 3 *Algorithm VarBatch is resource competitive for* $[\Delta \mid 1 \mid D_\ell \mid 1]$.

Proof. Consider any input σ for $[\Delta \mid 1 \mid D_\ell \mid 1]$. Suppose there exists an offline schedule S for σ with cost C and m resources. By Lemma 5.3, there exists a punctual schedule S' for σ with cost $O(C)$ and $O(m)$ resources. Let σ' be a request sequence obtained by delaying the arrival of each job x of delay bound p that arrives in $halfBlock(p,i)$ in σ until $halfBlock(p,i+1)$ and restricting the execution of x to $halfBlock(p,i+1)$. Since S' is punctual, there exists an offline schedule S'' for σ' that behaves exactly as S'.

The sequence σ' can be viewed as an input sequence for $[\Delta \mid 1 \mid \frac{D_\ell}{2} \mid \frac{D_\ell}{2}]$. By Theorem 1, algorithm Distribute is resource competitive for $[\Delta \mid 1 \mid \frac{D_\ell}{2} \mid \frac{D_\ell}{2}]$. Hence, algorithm Distribute generates an online schedule T for σ' that is resource competitive with S''. Therefore, T incurs cost $O(C)$ with $O(m)$ resources. For σ, algorithm VarBatch first transforms σ into σ' by delaying the job arrivals and then applies algorithm Distribute to generate schedule T for σ'. The schedule T is also the final schedule for σ.

In summary, for any input σ for $[\Delta \mid 1 \mid D_\ell \mid 1]$, if there exists an offline schedule S for σ with cost C and m resources, algorithm VarBatch generates a schedule T for σ with cost $O(C)$ and $O(m)$ resources. Therefore, algorithm VarBatch is resource competitive for $[\Delta \mid 1 \mid D_\ell \mid 1]$. ∎

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 $2^j \le p < 2^{j+1}$, and any job x with delay bound p that arrives in $halfBlock(2^{j-1}, i)$, we delay the arrival of x until $halfBlock(2^{j-1}, i+1)$ and restrict the execution of x in $halfBlock(2^{j-1}, i+1)$. The proof of the extended solution is similar as given in Section 5.2.

References

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

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

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

[4] 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.

[5] 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.

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

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

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

[9] 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.

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

[11] 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.

[12] 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.

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

[14] 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*, July/August 2006. To appear.

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

[16] 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.

[17] 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*, February 2003.

[18] 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.

A Analysis of ΔLRU

In this section, we show that ΔLRU is not constant competitive even with a nonconstant factor blowup in the number of resources.

Let OFF denote an arbitrary offline algorithm. We give OFF one resource (the argument can be easily extended to the general case that OFF has more than one resource). Recall that we give ΔLRU n resources. Consider $\frac{n}{2}$ colors with a delay bound 2^j and one color with a delay bound 2^k, where $2^k > 2^{j+1} > n\Delta$. For convenience, we refer to each color with a delay bound 2^j as a short-term color and the color with a delay bound 2^k as a long-term color. The input sequence proceeds in 2^k rounds as follows. We receive Δ jobs for each of the short-term color every integral of 2^j, and 2^k jobs for the long-term color at the very beginning.

It is not hard to verify that the timestamp of any short-term color is always at least as recent as that of the long-term color. Hence, ΔLRU caches all short-term colors at the beginning of the second integral of 2^j and then keeps the same configuration afterwards. The reconfiguration cost incurred by ΔLRU is $n\Delta$. The drop cost incurred by ΔLRU is at least 2^k.

Consider an offline algorithm OFF that caches the long-term color throughout. The reconfiguration cost incurred by OFF is Δ. The drop cost incurred by OFF is $2^{k-j-1}n\Delta$. Hence, the competitive ratio of ΔLRU is at least

$$\frac{n\Delta + 2^k}{\Delta + 2^{k-j-1}n\Delta}.$$

Because $2^k > 2^{j+1} > n\Delta$, we obtain that the competitive ratio is $\Omega(\frac{2^{j+1}}{n\Delta})$, which is not a constant when j is sufficiently large.

B Analysis of EDF

In this section, we show that EDF is not constant competitive even with a nonconstant factor blowup in the number of resources.

Let OFF denote an arbitrary offline algorithm. We give OFF one resource (the argument can be easily extended to the general case that OFF has more than one resource). Recall that we give EDF n resources. We consider $\frac{n}{2}+1$ colors as follows: a color with a delay bound 2^j, a color with a delay bound 2^k, a color with a delay bound 2^{k+1}, ..., and a color with a delay bound $2^{k+\frac{n}{2}-1}$, where $2^k > 2^j > \Delta > n$. The input sequence proceeds in $2^{k+\frac{n}{2}-1}$ rounds as follows. For the color with a delay bound of 2^j, we receive Δ jobs for each integral multiple of 2^j, until round 2^{k-1}. For a color with a delay bound of 2^{k+p}, for $0 \leq p < \frac{n}{2}$, we receive 2^{k+p-1} jobs at the very beginning.

For the above input sequence, EDF first caches the $\frac{n}{2}$ color with the smallest delay bounds, and then executes jobs for the color with the largest delay bound whenever any resource becomes idle. The reconfiguration cost incurred by EDF is at least $2^{k-j-1}\Delta$.

Consider an offline algorithm OFF that caches the color with a delay bound of 2^j throughout rounds from 0 to $2^{k-1}-1$, and caches the color with a delay bound of 2^{k+p} throughout rounds from 2^{k+p-1} to $2^{k+p}-1$, where $0 \leq p < \frac{n}{2}$. Algorithm OFF does not incur any drop cost and incurs a reconfiguration cost of $\left(\frac{n}{2}+1\right)\Delta$.

Hence the competitive ratio of EDF is at least $\frac{2^{k-j-1}}{\frac{n}{2}+1}$, which is not a constant if $k-j$ is sufficiently large.