Reconfigurable Resource Scheduling

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, mitul, vin}@cs.utexas.edu

ABSTRACT

We consider a class of scheduling problems that we refer to as reconfigurable resource scheduling. This class of problems is motivated by emerging applications that involve dynamically allocating a large number of shared resources to a variety of services. We design efficient online algorithms for certain problems in this class. Our goal is to obtain constant competitive online algorithms where the online algorithm is given a constant factor advantage in terms of the number of resources. The main problem considered in this paper is as follows. 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 processed within a fixed delay bound from its arrival, or else it is dropped and we incur a category-specific 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. Our main result is a constant competitive online algorithm for this problem, which is obtained by the following layered approach. First, we reduce our main problem to the special case in which all jobs arrive at integral multiples of the delay bound. Second, we reduce the latter problem to the special case of unit delay. Third, we reduce the unit-delay problem to a caching problem that we refer to as file caching with remote reads. Our solution to this caching problem generalizes certain existing work in the area of file caching.

Categories and Subject Descriptors

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

General Terms

Algorithm, Performance

[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.

SPAA'06, July 30–August 2, 2006, Cambridge, Massachusetts, USA.

Keywords

Online computation, reconfigurable resource scheduling

1. INTRODUCTION

In emerging networked systems, it is common that resources are shared by a variety of services or applications. For example, a shared data center [10, 11] is a cluster of machines managed to provide multiple independent services. As another example, emerging packet processing systems based on programmable, multi-core network processors [1, 2, 19, 22] share cores among various applications. In both of these examples, the loads on the different services fluctuate with time in an unpredictable manner. A static partition of the resources often fails to provide a satisfactory performance guarantee. On the other hand, over-provisioning, that is, providing a sufficient number of resources to meet the peak load, can lead to underutilization of the resources. A desirable alternative is to dynamically adjust the resource allocation.

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

In this paper, we initiate the study of reconfigurable resource scheduling within the framework of competitive analysis (see the textbook by Borodin et al. [6] for a comprehensive introduction of competitive analysis) by considering the following specific problem in this realm. We are given a finite set of resources, each of which has an associated color, and a sequence of requests, each of which is a set of unit jobs. Each job has an associated color, and needs to be executed on a resource of the same color within a fixed delay bound D from its arrival, or else it is dropped. At any time, a resource can be reconfigured to a different color. The cost to reconfigure a resource is Δ, and the cost to drop a job of color ℓ is d_ℓ. 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 paper, we give a constant competitive online algorithm for this problem, where we allow the online algorithm a constant factor advantage in the number of resources. It is worth mentioning that the competitive ratio we obtain does not depend on the various problem parameters, that is, D, Δ, and d_ℓ's.

We solve the main 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. Second, we reduce the latter problem to two cases: (1) $\Delta < d_\ell$, for all ℓ, and (2) $\Delta \geq d_\ell$, for all ℓ. We solve the first case using a relatively simpler approach. For the second case, we use a *reshaping* technique that enables us to reduce to the special case in which $D = 1$. For the case $D = 1$, we actually solve a more general variation that allows a per-color reconfiguration cost Δ_ℓ, as long as $\Delta_\ell \geq d_\ell$ for all ℓ. Third, we use a *serialization* technique to reduce the case $D = 1$ to a caching problem that we refer to as file caching with remote reads. This caching problem generalizes the file caching problem studied by Cao and Irani [9] and Young [23], and we solve it by modifying Young's Landlord algorithm.

2. RELATED WORK

Brucker [7, 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. [8] 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. [20] 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. In [13], Kokku proposes 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. 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 interesting related scheduling problem is "scheduling with rejection" [5, 16, 17]. 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.

A traffic regulator like leaky bucket [21] reduces the burstiness in the network traffic. In this paper, we use a reshaping technique to distribute each job to a specific round, which can be viewed as a way to reduce the burstiness in the request sequence.

As indicated earlier, we generalize the file caching work of Cao and Irani [9] and Young [23], which themselves can be viewed as generalizations of the work in the classic disk paging problem studied by Sleator and Tarjan [18]. In the latter paper, which is the first work in the area of competitive analysis, certain algorithms, such as LRU, are shown to be constant competitive when given a constant factor advantage in the cache size. The technique of giving the online algorithm extra resources to achieve a better competitive ratio has subsequently been referred to as resource augmentation [12, 15].

Some other work related to our paging problem includes k-server problem with excursions and page migration problem. Manasse et al. [14] consider k-server problem with excursions in which, a request can be satisfied remotely by servers without moving any server to the requested vertex. In page migration related problems, extensively studied in the literature, the requested page can be accessed remotely or moved to the requesting processor. Some closely related work in this realm are k-page migration [4] and constrained page migration [3]. In [4], Bartal et al. maintain k copies of any page, however, local memories have unlimited capacities. In [3], Albers and Koga consider page migration with limited local memory capacity. There are some similarities between their DLRU algorithm and our file caching algorithm presented in Section 4.2. However, DLRU algorithm is not applicable directly to solve our file caching problem.

3. PRELIMINARIES

In this section, we give problem definitions in Section 3.1 and the organization of the rest of the paper in Section 3.2.

3.1 Problem Definitions

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. After a job arrives, it is *pending* until it starts to get executed or dropped.

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. 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. There is a drop cost to drop a request and a reconfiguration cost to reconfigure a resource. The total cost of a schedule is the sum of the total reconfiguration and drop cost. 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 on 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.

The focus of this paper is to give resource competitive online algorithms for some problems in the class of reconfigurable resource scheduling. For brevity of the presentation, we introduce the [*reconfig* | *drop* | *delay* | *batch*] notation. The *reconfig* field describes the details of the reconfiguration cost. In this paper, the possible values for this field are: a fixed reconfiguration cost denoted by Δ and a per-color reconfiguration cost denoted by Δ_ℓ. The *drop* field describes the details of the drop cost. In this paper, there is only one possible value: a per-color drop cost denoted by d_ℓ. The *delay* field contains the details of the delay bound. In this paper, the possible values are: a unit delay, namely 1, and a fixed delay denoted by D. 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.

With this notation, our main problem is denoted by $[\Delta \mid d_\ell \mid D \mid 1]$. The special case in which jobs arrive at integral multiples of D is denoted by $[\Delta \mid d_\ell \mid D \mid D]$. The special case with $D = 1$ is denoted by $[\Delta \mid d_\ell \mid 1 \mid 1]$.

3.2 Roadmap

The rest of the paper is organized as follows. Section 4 defines and solves the problem file caching with remote reads. Section 5 solves $[\Delta_\ell \mid d_\ell \mid 1 \mid 1]$, where $\Delta_\ell \geq d_\ell$, by a reduction to file caching with remote reads. Section 6 solves $[\Delta \mid d_\ell \mid D \mid D]$ by reducing to special cases of $[\Delta \mid d_\ell \mid 1 \mid 1]$. Section 7 solves our main problem $[\Delta \mid d_\ell \mid D \mid 1]$ by a reduction to $[\Delta \mid d_\ell \mid D \mid D]$.

4. 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 of the final solution to our main problem. This problem is similar to the file caching problem studied by Cao and Irani [9] and Young [23]. The difference is that, on a miss, a remote read can be issued to serve the request instead of having to write the requested file to the cache. We modify the Landlord algorithm and its associated analysis given by Young to solve our caching problem.

4.1 Problem Definition

We are given a universal set of files and a cache of a certain size. Each file x is characterized by a positive integer size, denoted by $\mathit{size}(x)$; a nonnegative read cost, denoted by $\mathit{read}(x)$; a nonnegative write cost, denoted by $\mathit{write}(x)$. Initially, the cache is empty. The input is a sequence of requests, each of which is specified by a file (the file to be accessed). 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 $\mathit{write}(x)$, 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 $\mathit{read}(x)$. The goal is to maintain the files in the cache so as to minimize the total cost.

4.2 Algorithm

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

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

When $\mathit{credit}(x)$ reaches $\mathit{write}(x)$, 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 $\mathit{credit}(x)$ 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.

4.3 Analysis

Before presenting the analysis, let us first introduce some definitions. Let OFF denote an arbitrary offline algorithm. Let A and L denote the cache of OFF and LLL, respectively. Let m and n ($n > 2m$) denote the size of A and L, respectively. We define a potential function

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

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

To analyze the performance of LLL, we execute LLL alongside OFF. As in [23], 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 4.1 through Lemma 4.6.

LEMMA 4.1. *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 contents of A as well as, for any file x, $\mathit{credit}(x)$ do not change, Φ remains unchanged. □

LEMMA 4.2. *If OFF writes x to the cache, Φ increases by at most $(n - m + 1) \cdot \mathit{write}(x)$.*

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

LEMMA 4.3. *If LLL augments the credit of x that is not in A, Φ increases by at most $m \cdot read(x)$.*

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

LEMMA 4.4. *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 because $write(x) \geq credit(x)$. Hence, Φ does not increase. $\square$

LEMMA 4.5. *If LLL augments the credit of x that is in A, Φ decreases by at least $(n-2m+1)s \geq 0$, where $s \leq read(x)$. Particularly, 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 $s \leq read(x)$. Particularly, if $s < read(x)$, after the credit augmentation, $credit(x)$ reaches $write(x)$. LLL subsequently write x into the cache and does not perform a remote read in serving x. Because $x \in A$, the second summation decreases by $(n-m+1)s$. Hence, Φ decreases by at least $(n-2m+1)s \geq 0$. $\square$

LEMMA 4.6. *If LLL charges rent to make room for x, Φ does not increase.*

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

Consider any request sequence σ. Let $Cost_{OFF}(\sigma)$ and $Cost_{LLL}(\sigma)$ denote the cost incurred by OFF and LLL on σ, respectively. Let $ReadCost_{OFF}(\sigma)$ and $ReadCost_{LLL}(\sigma)$ denote the read cost incurred by OFF and LLL in serving σ, respectively. Let $WriteCost_{OFF}(\sigma)$ and $WriteCost_{LLL}(\sigma)$ denote the write cost incurred by OFF and LLL in serving σ, respectively.

LEMMA 4.7. *For any request sequence σ, the total increase of Φ is at most*

$$m \cdot ReadCost_{OFF}(\sigma) + (n-m+1)\, WriteCost_{OFF}(\sigma).$$

PROOF. Consider the steps taken by OFF and LLL to serve a request x. By Lemma 4.1 through Lemma 4.6, Φ increases only in the following two cases. The first case, OFF writes x to the cache. By Lemma 4.2, Φ increases by at most $(n-m+1)write(x)$. In this case, the write cost incurred by OFF in serving x is at least $write(x)$. The second case, LLL updates the credit of x that is not in A. By Lemma 4.3, Φ increases by at most $m \cdot read(x)$. In this case, the read cost incurred by OFF in serving x is $read(x)$. In either case, the increase of Φ in serving x is at most $m \cdot ReadCost_{OFF}(x) + (n-m+1)\, WriteCost_{OFF}(x)$. Summing up over all x's, the lemma follows. $\square$

LEMMA 4.8. *For any request sequence σ, the total negative change of Φ is at least*

$$(n-2m+1)(ReadCost_{LLL}(\sigma) - ReadCost_{OFF}(\sigma)).$$

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 $(n-2m+1)read(x)$ by Lemma 4.5; otherwise, OFF incurs a read cost of $read(x)$ in serving x. In either case, $(n-2m+1)ReadCost_{LLL}(x)$ is at most the decrease of Φ in serving x plus $(n-2m+1)ReadCost_{OFF}(x)$, so the decrease of Φ in serving x is at least $(n-2m+1)(ReadCost_{LLL}(x) - ReadCost_{OFF}(x))$. Summing up over all such x's, the lemma follows. $\square$

LEMMA 4.9. *For any request sequence σ,*

$$WriteCost_{LLL}(\sigma) \leq ReadCost_{LLL}(\sigma).$$

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 copy 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 $write(x)$, for each access on x, the credit increases by at most $read(x)$, and algorithm LLL incurs a read cost of $read(x)$. When the credit reaches $write(x)$, algorithm LLL writes x into the cache, incurring a write cost of $write(x)$. After that, 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 i's and files, the lemma follows. $\square$

THEOREM 1. *Algorithm LLL is $\frac{2(n-m+1)}{n-2m+1}$-competitive.*

PROOF. Consider an arbitrary request sequence σ. By Lemma 4.7, Lemma 4.8 and the fact that Φ is always nonnegative, we have

$$\begin{aligned} & (n-2m+1)(ReadCost_{LLL}(\sigma) - ReadCost_{OFF}(\sigma)) \\ \leq\ & m \cdot ReadCost_{OFF}(\sigma) + (n-m+1)\, WriteCost_{OFF}(\sigma). \end{aligned}$$

Because $n > 2m$, we obtain that

$$ReadCost_{LLL}(\sigma) \leq \frac{n-m+1}{n-2m+1} Cost_{OFF}(\sigma).$$

The lemma follows from the above inequality and Lemma 4.9. $\square$

5. UNIT DELAY

In this section we solve $[\Delta_\ell \mid d_\ell \mid 1 \mid 1]$, where $\Delta_\ell \geq d_\ell$ for all ℓ. Recall that it is characterized by a per-color configuration cost Δ, a per-color drop cost 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 4.

5.1 Algorithm Serialize

Algorithm Serialize proceeds in three steps. In the first step, given an arbitrary instance I of $[\Delta_\ell \mid d_\ell \mid 1 \mid 1]$, we construct an instance I' of file caching with remote reads as follows. The cache size associated with I' is the same as the number of resources associated with I. Each file (ℓ, j) associated with I' is characterized by a color ℓ, a nonnegative integer index j, a read cost d_ℓ, and a write cost Δ_ℓ. Let σ be the input sequence associated with I. For any nonnegative integer i, let σ_i be request i of σ. Let $X_i = \cup_\ell \{(\ell, j) \mid 0 \leq$

$j < q_{i,\ell}\}$, where $q_{i,\ell}$ is the number of color ℓ jobs in σ_i. Let σ'_i be the sequence of requests obtained by ordering the files in X_i arbitrarily. The input sequence σ' associated with I' is obtained by concatenating σ'_i's in increasing order of i.

In the second step, we use algorithm LLL (defined in Section 4.2) to obtain a solution S' for I'. In the third step, we construct a solution S for I from S' in two substeps. In the first substep, we construct another solution S'' for I' as follows. Let α_i be the schedule for σ'_i in S'. We obtain β_i, the schedule of S'' for σ'_i, by delaying the writes in α_i to the end of α_i. The schedule S'' is the concatenation of β_i's in increasing order of i. In the second substep, we construct a solution S for I as follows. Fix an arbitrary i. Now we describe the schedule of S in round i. For any k, S configures resource k with color ℓ in round i, where ℓ is the color of the file cached by S'' in slot k at the beginning of σ'_i. In round i, S executes as many jobs in σ_i as its configuration allows.

5.2 Analysis

LEMMA 5.1. *If there exists an offline solution T for I with cost C and m resources, then there exists an offline solution T' for I' with cost at most $4C$ and a cache of size m.*

PROOF. We construct T' round by round. Fix an arbitrary round i. For any color ℓ, let $X_{i,\ell}$ be the set of resources configured with color ℓ by T in round i. The schedule of T' for σ'_i is constructed in the following two steps.

First, in T, we label the resources in round i. For each color ℓ, we label the resources in $X_{i,\ell}$ in round i as follows. If $i = 0$, we label the resources in $X_{i,\ell}$ from 0 to $|X_{i,\ell}| - 1$ arbitrarily. If $i > 0$, we proceed as follows. For any resource k in $X_{i,\ell}$ that is configured with color ℓ in round $i-1$, such that the label of resource k in round $i-1$ is in the range $[0, |X_{i,\ell}|)$, we let resource k inherit its label from round $i-1$. For any remaining resource k in $X_{i,\ell}$, we assign to k an arbitrary label in the range $[0, |X_{i,\ell}|)$ that is not already assigned to a resource in $X_{i,\ell}$ in round i.

Second, we construct the schedule of T' for σ'_i from the labels given to resources in round i in the first step. Throughout σ'_i, we maintain the following cache contents: for any nonnegative integer k, the page cached at location k is (ℓ, j), where ℓ is the color of resource k in round i of T, and j is the label given to resource k in round i. (An exception occurs if the color of resource k is black, in which case cache location k is empty.)

We need to show that (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' is at most the drop cost incurred by T.

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

In the second step, we need to show that the write cost incurred by T' is at most twice the total credit. Since T' maintains the same cache contents throughout each σ'_i, T' only incurs write cost on the boundaries of σ'_i's. Hence, it is sufficient to show that, for any positive integer i, the write cost incurred by T' on the boundary between σ'_{i-1} and σ'_i is at most twice the credit associated with the reconfigurations incurred by T between round $i-1$ and round i.

Fix any nonnegative integer k. Consider any write in T' on the boundary between σ'_{i-1} and σ'_i at location k, because of a reconfiguration in T between round $(i-1)$ and round i on resource k. It is easy to see that the write cost is at most the credit associated with the reconfiguration.

Now consider writes in T' because of 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 from round $i-1$ to round i in T. Let $q_{i,\ell}$ be the number of resources that are configured with color ℓ in both round $i-1$ and round i in T, but changes labels from round $i-1$ to round i. It is easy to verify that the write cost incurred by T' on the boundary between σ'_{i-1} and σ'_i due to 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 incurred by T between round $i-1$ and round i is at least $\sum_\ell p_{i,\ell} \cdot \Delta_\ell$. Hence, we only need to show that $q_{i,\ell} \le p_{i,\ell}$, which is shown as follows.

Let $Y_{i,\ell}$ be the set of resources that are configured with color ℓ and each of which has a label at least $|X_{i+1,\ell}|$ in round i. By the way we assign labels to resources in each round, $q_{i,\ell}$ equals $|Y_{i-1,\ell}|$. It is not hard to see that $|Y_{i-1,\ell}|$ is at most $\max(0, |X_{i-1,\ell}| - |X_{i,\ell}|)$, which in turn is at most $p_{i,\ell}$. Hence, $q_{i,\ell} \le p_{i,\ell}$. Therefore, the write cost incurred by T' on the boundary between σ'_{i-1} and σ'_i is at most twice the total credits associated with reconfigurations in T.

The proof of (2) proceeds as follows. Fix arbitrary ℓ and i. Let $r_{i,\ell}$ be the number of color ℓ jobs in σ_i. Let $k_{i,\ell}$ is the number of resources configured with color ℓ by T in round i. So in round i, T pays a drop cost of at least $d_\ell \cdot \max(r_{i,\ell} - k_{i,\ell}, 0)$ on the color ℓ jobs in σ_i. From the way σ' is constructed, the set of color ℓ files in σ'_i is $\{(\ell, j) \mid 0 \le j < r_{i,\ell}\}$. From the way T' is constructed, throughout σ'_i, the set of color ℓ files cached by T' is $\{(\ell, j) \mid 0 \le j < k_{i,\ell}\}$. So T' pays a read cost of $d_\ell \cdot \max(r_{i,\ell} - k_{i,\ell}, 0)$ on color ℓ files 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 ℓ's and i's, the claim follows. ☐

LEMMA 5.2. *The cost incurred by S is at most twice that incurred by S'.*

PROOF. First, we establish that the cost incurred by S'' is at most twice that incurred by S' as follows. Fix an arbitrary i. Recall that α_i and β_i are the schedules of S' and S'' for σ'_i, respectively. Because β_i is obtained by delaying writes in α_i, the write cost incurred by S'' in β_i is the same as that incurred by S' in α_i, and for files that are not written in α_i, the read cost incurred by S'' are the same as that incurred by S'. From the way σ'_i is constructed, each file in σ'_i is unique. Let X be the set of files written by S' in α_i. For each file (ℓ, j) in X, S'' incurs at most one read in β_i. Because $d_\ell \le \Delta_\ell$, the read cost incurred by S'' on X in β_i is at most the write cost incurred by S' on X in α_i. Hence, the cost incurred by S'' on σ'_i is at most twice the cost incurred by S' on σ'_i. Summing up over all i's, the claim follows.

Second, we show that the cost incurred by S is at most that incurred by S''. To do this, we first show that the reconfiguration cost incurred by S is at most the write cost incurred by S''. It is not hard to verify that each reconfiguration of a resource to color ℓ performed by S corresponds to a write of a color ℓ file by S''. The claim then follows from the fact that the cost to reconfigure a resource to color ℓ and to write a color ℓ file are both Δ_ℓ. We then establish that the drop cost incurred by S is the same as the read cost incurred by S''. Fix arbitrary ℓ and i. Let $r_{i,\ell}$ be the number of color ℓ jobs in σ_i. From the way σ'_i is constructed, the set of color ℓ files in σ'_i is $\{(\ell, j) \mid 0 \le j < r_{i,\ell}\}$. Let $k_{i,\ell}$ be the number of color ℓ files cached by S'' at the beginning of σ'_i. From the way β_i is constructed, S'' does not change the cache contents during β_i. So the read cost incurred by S'' on color ℓ files in σ'_i is at least $d_\ell \cdot \max(r_{i,\ell} - k_{i,\ell}, 0)$. By the way S is constructed, the number of resources configured with color ℓ in round i is also $k_{i,\ell}$, and S executes as many jobs as possible. So the drop cost incurred by S on color ℓ jobs in σ_i is $d_\ell \cdot \max(r_{i,\ell} - k_{i,\ell}, 0)$. Hence, the drop cost incurred by S on color ℓ jobs in σ_i is at most the read cost incurred by S'' on color ℓ files in σ'_i. Summing up over all ℓ's and i's, the claim follows. □

THEOREM 2. *Algorithm Serialize is resource competitive for $[\Delta_\ell \mid d_\ell \mid 1 \mid 1]$, where $\Delta_\ell \ge d_\ell$ for all ℓ.*

PROOF. Suppose there exists an offline solution T for I with cost C and m resources. By Lemma 5.1, there exists an offline solution T' for I' with at most cost $4C$ and a cache of size m. By Theorem 1, given a constant factor advantage in the number of resources, algorithm LLL is constant competitive. Hence S', the solution given by LLL for I', incurs a cost of $O(C)$ with a cache size of $O(m)$. By Lemma 5.2 and the fact that the number of resources given to S is the same as the cache size given to S', we obtain that S incurs a cost of $O(C)$ with $O(m)$ resources. □

6. BATCHED ARRIVALS

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

In this section, we reduce the $[\Delta \mid d_\ell \mid D \mid D]$ to two cases: (1) $\Delta \ge d_\ell$, for all ℓ, and (2) $\Delta < d_\ell$, for all ℓ. We solve the former case by a reduction to $[\Delta \mid d_\ell \mid 1 \mid 1]$, which is addressed in Section 5. The latter case is relatively simpler. We solve the latter case to 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 Appendix B.

6.1 Definitions

For any i, we define block i to be the D rounds starting from round $i \cdot D$. In the following, we focus our attention on an arbitrary block i and the jobs that arrive in block i.

Let $load(\ell)$ denote the number of color ℓ jobs. For convenience, we number the jobs of color ℓ from 0 to $load(\ell) - 1$. We define groups as follows. A group is a set of jobs, identified by a pair (ℓ, j), where ℓ is a color and j is a nonnegative integer. A group (ℓ, j) consists of the following jobs of color ℓ: job $j \cdot D$, job $j \cdot D+1$, …, and job $\min((j+1)D, load(\ell))-1$, where $0 \le j < \left\lceil \frac{load(\ell)}{D} \right\rceil$.

A group $U = (\ell, j)$ is *heavy* if $|U| \cdot d_\ell \ge \Delta$, and *light* otherwise. Any job x in a heavy (resp., light) group is referred to as a heavy (resp., light) job. We denote the number of heavy groups by h. For convenience, we sort colors in descending order of per-color drop cost, breaking ties arbitrarily. We then sort heavy groups in descending order of per-color drop cost, breaking ties by color, and then by cardinality. We sort light groups in the same way. We denote heavy group j in the sorted order by H_j.

We denote the number of resources by n. We define n bins indexed from 0 to $n-1$. Each bin has D slots numbered from zero, where slot j corresponds to round $j + i \cdot D$. A slot is either *free* or *claimed*. At any instant, we denote the number of free slots in bin k by $free(k)$. Initially, all slots are free. A slot can become claimed only in the *distribute* procedure, which is defined as follows. Given two parameters, a set of jobs X and an integer k, where $0 \le k < n$ and $|X| \le free(k)$, the procedure *distribute* maps all jobs in X to the $|X|$ lowest numbered free slots of bin k such that one job is mapped to each slot. A slot becomes claimed when a job is assigned to it.

6.2 Algorithm

In this section, we first introduce algorithm Reshape in Section 6.2.1. After that, we give algorithm Split, which uses Reshape as a subroutine. Algorithm Split solves $[\Delta \mid d_\ell \mid D \mid D]$.

6.2.1 Algorithm Reshape

Algorithm Reshape takes an input sequence σ for $[\Delta \mid d_\ell \mid D \mid D]$ as a parameter, and maps each job in σ to a round. In other words, Reshape transforms an input sequence for $[\Delta \mid d_\ell \mid D \mid D]$ to an input sequence for $[\Delta \mid d_\ell \mid 1 \mid 1]$ by restricting each job to a certain round. Algorithm Reshape proceeds block by block. Fix any nonnegative integer i. We focus on the jobs that arrive in block i. Algorithm Reshape for block i proceeds as follows.

1. Empty all bins.
2. For each j such that $0 \le j < \min(n, h)$, we map the jobs in H_j by invoking *distribute*(H_j, j).
3. Let $j = n$, $k = 0$, and B_0, …, B_{n-1} be the set of bins in descending order of the number of free slots.
4. Let $X = H_j$. Let ℓ be the color of X, and p be the index of bin B_k.
5. If $d_\ell \cdot free(p) \ge \Delta$,
   (a) If $|X| \le free(p)$, we perform the following steps.
      i. We map the jobs in X by invoking *distribute*(X, p), and then increment j.
      ii. If $j \ge h$, go to 6.
      iii. We set X to H_j, and ℓ to the color of X.
   (b) Otherwise, we map the jobs in Y by invoking *distribute*(Y, p), and then we set X to $X \backslash Y$, where Y is any subset of X of size $free(p)$.
   (c) $free(p) = 0$, we perform the following steps.
      i. We increment k.
      ii. If $k \ge n$, go to 6.
      iii. Set p to be the index of B_k.
      iv. If $d_\ell \cdot free(p) < \Delta$, go to 6.
   (d) Repeat 5a, 5b, and 5c.
6. For each group U with at least one job not mapped, we map the remaining jobs in U to the beginning of block i, such that one job is mapped to one round.

6.2.2 Algorithm Split

Algorithm Splitis defined as follows. Consider any input sequence σ for $[\Delta \mid d_\ell \mid D \mid D]$. We break σ into two subsequences α and β, where α only consists of the jobs with drop cost at most Δ, and β consists of the remaining jobs. We split the resources into two equal halves, where the first half is used to execute only jobs in α, and the second half is used to execute only jobs in β.

The schedule for α is obtained in the following two steps. First, we construct an input α' by applying algorithm Reshape on α. Second, we apply algorithm Serialize (defined in Section 5.1) on α' to determine the schedule.

The schedule for α is obtained in the following two steps. First, we construct an input β' by applying algorithm Reshape on β. Second, we apply algorithm RL-Serialize (defined in Appendix B.1) on β' to determine the schedule.

6.3 Analysis

We define a resource to be *i-monochromatic* if the resource k is configured with one color throughout block i, and *i-multichromatic* otherwise. An i-monochromatic resource is (i, ℓ)-monochromatic if the resource is configured with color ℓ throughout block i. We assume that, for each i-multichromatic resource, there are reconfigurations at the beginning and end of block i. It is not hard to see that this assumption at most triples the reconfiguration cost. A reconfiguration is called an *external* reconfiguration if it is made on the delay bound boundary, otherwise it is an *internal* reconfiguration.

A schedule S is *reshape-friendly* if, in S, any job x is either dropped, or executed in the round to which x is mapped to by algorithm Reshape.

LEMMA 6.1. *For any input sequence σ and any schedule S for σ with cost C, there exists a reshape-friendly schedule S' for σ with cost $O(C)$.*

Proof sketch. Given any schedule S, we sketch how to transform it to a reshape-friendly schedule S'. The transformation proceeds block by block. In the following we focus on an arbitrary block i and the jobs that arrive in block i.

At a high level, the transformation for block i proceeds in five steps as follows. First, we rearrange the schedule so that jobs of the same color on the same resource are executed in consecutive rounds. Second, we perform a labeling process that assigns jobs to groups (we will elaborate this process later). Third, we perform a unifying process that brings together the jobs in each heavy group H_j, for $0 \le j < \min(n, h)$ (we will give the details of this process later). Fourth, we drop light jobs on i-multichromatic resources. Fifth, if $n > h$, we perform a repacking process, which reschedules the jobs in $\{H_j \mid n \le j < h\}$ in the same way as they are reshaped by algorithm Reshape. In general, each step preserves the properties obtained from previous steps and adds more structures to the schedule. The first, fourth, and fifth steps are straightforward. In the following we present the details of the remaining steps .

The purpose of the second step, that is, the labeling process, is to assign jobs to groups in a way that avoids some extra reconfigurations that might otherwise be caused by the unifying process. It works as follows. For each color ℓ, we scan schedules of resources in ascending order of resource indices and then the dropped jobs in an arbitrary order to obtain a sequence α_ℓ of color ℓ jobs. We number the jobs in α_ℓ from zero. For any color ℓ and any j such that $0 \le j < \left\lceil \frac{load(\ell)}{D} \right\rceil$, we assign the following jobs of α_ℓ to group (ℓ, j): job $j \cdot D$, job $j \cdot D + 1$, …, and job $\min((j+1)D - 1, load(\ell))$.

The unifying process proceeds in sorted order of heavy groups. Now we describe the unifying process wrt. an arbitrary heavy group (ℓ, j) in the following two steps. First, we pick a resource k on which we will unify jobs in group (ℓ, j) as follows. Let P be the set of resources not previously picked by the unifying process. If X is empty, we terminate the unifying process. Otherwise, (1) if there exists a resource that schedules at least one job in group (ℓ, j), pick k to be the smallest index of such a resource; (2) otherwise, if there exists an i-multichromatic resource, pick k to be the index of such a resource; (3) otherwise, if there exists an i-monochromatic resource, pick k to be the index of such a resource configured with color ℓ' such that $d_{\ell'}$ is the smallest; (4) otherwise, terminate the unifying process. Second, we unify the jobs in group (ℓ, j) on resource k in three substeps as follows. In the first substep, we shift the jobs in (ℓ, j) scheduled on resource k to the beginning of block i. Other jobs on resource k are shifted towards the end of block i. In the second substep, for each remaining resource p, if there are jobs in (ℓ, j) scheduled on p, transfer them to resource k via swapping to ensure that all jobs in (ℓ, j) are scheduled continuously on resource k. In the third substep, execute a set of jobs X on resource k via swapping in the same fashion as in the second substep, where X is the set of jobs in group (ℓ, j) that are dropped by S. The jobs swapped out are dropped.

We need to show that (1) S' is reshape-friendly and (2) the transformation does not increase the cost by more than a constant factor. The proof of the former turns out to be straightforward. In the following we give a high level idea of the cost accounting. The first and fourth steps do not increase cost. The second step does not incur any cost. To account for the cost increase in the third step, that is, the unifying process, we associated $O(d_\ell)$ units of credit with each drop incurred by S, and $O(\Delta)$ units of credit with each reconfiguration incurred by S. We can then show that the increased cost incurred by the unifying process can be paid for by the total credit associated with the drops and reconfigurations incurred by S. To account for the cost increase in the fifth step, that is, the repacking process, we can first show that, if jobs dropped by S are omitted from each group, the cost increase incurred by the repacking process is at most a constant factor of that incurred by S. We can then show a similar result if jobs dropped by S are present in each group. This completes the proof sketch. $\square$

LEMMA 6.2. *Algorithm Split is resource competitive for $[\Delta \mid d_\ell \mid D \mid D]$, where $\Delta \ge d_\ell$, for all ℓ.*

PROOF. Consider any input σ for $[\Delta \mid d_\ell \mid D \mid D]$, where $\Delta \ge d_\ell$, for all ℓ. Suppose there exists an offline schedule S for σ with cost C and m resources. By Lemma 6.1, there exists a reshape-friendly schedule S' for σ with cost $O(C)$ and $O(m)$ resources. Let σ' be a request sequence obtained by applying algorithm Reshape on σ. Because S' is reshape-friendly, there exists an offline schedule S'' for σ' that behaves exactly as S'.

The sequence σ' can be viewed as a input sequence for

$[\Delta \mid d_\ell \mid 1 \mid 1]$. By Theorem 2, algorithm Serialize is resource competitive for $[\Delta \mid d_\ell \mid 1 \mid 1]$. Hence, algorithm Serialize 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 Split first transforms σ into σ' using algorithm Reshape and then applies Serialize to generate schedule T for σ'. The schedule T is also the final schedule for σ.

In summary, if there exists an offline schedule S for σ with cost C and m resources, then algorithm Split generates a schedule T for σ with cost $O(C)$ and $O(m)$ resources. From this claim, together with the fact that algorithm Split uses half the resources to generate schedule T for σ, the lemma follows. □

The proof for Lemma 6.3 is analogous to the proof for Lemma 6.2. In fact, the proof for Lemma 6.3 can be obtained by replacing $\Delta \geq d_\ell$ with $\Delta < d_\ell$, $[\Delta \mid d_\ell \mid 1 \mid 1]$ with rate-limited $[\Delta \mid d_\ell \mid 1 \mid 1]$ (see Appendix B), Serialize with RL-Serialize (see Appendix B.1), and Theorem 2 with Theorem 5 in the proof for Lemma 6.2.

LEMMA 6.3. *Algorithm Split is resource competitive for* $[\Delta \mid d_\ell \mid D \mid D]$*, where* $\Delta < d_\ell$*, for all* ℓ*.*

THEOREM 3. *Algorithm Split is resource competitive for* $[\Delta \mid d_\ell \mid D \mid D]$.

PROOF. For any input sequence σ for $[\Delta \mid d_\ell \mid D \mid D]$, we partition σ into two subsequences α and β, where α consists of the jobs with drop cost at most Δ, and β consists of the remaining jobs. Let S be any offline schedule for σ. Let S_α (resp., S_β) be the schedule of S for α (resp., β). Obviously, the cost incurred by each of S_α and S_β is at most that incurred by S.

Let T be the scheduled generated by algorithm Split for σ. Let T_α (resp., T_β) be the schedule of T for α (resp., β). Since Split dedicates one half of the resources to α and the other half to β, the cost incurred by T is the sum of the cost incurred by T_α and T_β, respectively.

By Lemma 6.2, T_α is resource competitive with S_α. By Lemma 6.3, T_β is resource competitive with S_β. Hence, T is resource competitive with S, and the theorem follows. □

7. OUR MAIN RESULT

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, a per-color drop cost d_ℓ, a fixed drop cost D, and nonbatched arrivals (request can arrive at 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 6.

7.1 Definitions

A half-block is defined as follows. Half-block i is the $\frac{D}{2}$ rounds starting from round $\frac{i \cdot D}{2}$. Consider any input sequence σ and any algorithm A. For any job x executed by A, let i be the index of the half-block in which x arrives. We say the execution of x is *early* if x is executed in half-block i, *punctual* if executed in half-block $i+1$, and *late* if executed in half-block $i+2$.

7.2 Algorithm Batch

Algorithm Batch is as follows. Given any input σ for $[\Delta \mid d_\ell \mid D \mid 1]$, we first construct an input σ' for $[\Delta \mid d_\ell \mid \frac{D}{2} \mid \frac{D}{2}]$ by delaying any job x that arrives in half-block i in σ until half-block $i+1$, and we then apply algorithm Split (defined in Section 6.2.2) on σ' to obtain the schedule.

7.3 Analysis

We define a schedule S to be *batch-friendly* if all job executions in S are punctual.

LEMMA 7.1. *Consider any input* σ *for* $[\Delta \mid d_\ell \mid D \mid 1]$*. If there exists an offline schedule* S *for* σ *with cost* C *and* m *resources, then there exists a batch-friendly offline schedule* S' *for* σ *with cost* $O(C)$ *and* $O(m)$ *resources.*

PROOF. The schedule S' is constructed as follows. We give $3m$ resources to S'. We use the first m resources of S' to schedule only jobs whose executions are early in S, where each reconfiguration and each early execution performed by S are postponed by $\frac{D}{2}$ rounds. We use the second m resources of S' to schedule only jobs whose executions are punctual in S, where each reconfiguration and each punctual execution performed by S are performed in the same round as in S. We use the third m resources of S' to schedule only jobs whose executions are late in S, where each reconfiguration (except those made in the first half-block by S) and each late execution performed by S are performed $\frac{D}{2}$ rounds earlier.

In each of the three m resources, S' incurs at most the reconfiguration cost as that incurred by S. Hence, the total reconfiguration cost incurred by S' is at most three times that incurred by S. It is not hard to see each job execution in S becomes a punctual execution in S'. Hence, S' is batch-friendly and incurs at most the same drop cost as that incurred by S. Therefore, the lemma follows. □

THEOREM 4. *Algorithm Batch is resource competitive for* $[\Delta \mid d_\ell \mid D \mid 1]$.

PROOF. Consider any input σ for $[\Delta \mid d_\ell \mid D \mid 1]$. Suppose there exists an offline schedule S for σ with cost C and m resources. By Lemma 7.1, there exists a batch-friendly 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 that arrives in half-block i in σ until half-block $i+1$. Because S' is batch-friendly, there exists an offline schedule S'' for σ' that behaves exactly as S'.

The sequence σ' can be viewed as a input sequence for $[\Delta \mid d_\ell \mid \frac{D}{2} \mid \frac{D}{2}]$. By Theorem 3, algorithm Split is resource competitive for $[\Delta \mid d_\ell \mid \frac{D}{2} \mid \frac{D}{2}]$. Hence, algorithm Split 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 Batch first transforms σ into σ' by delaying the job arrivals and then applies algorithm Split to generate schedule T for σ'. The schedule T is also the final schedule for σ.

In summary, if there exists an offline schedule S for σ with cost C and m resources, algorithm Batch generates a schedule T for σ with cost $O(C)$ and $O(m)$ resources. Therefore, algorithm Batch is resource competitive. □

8. REFERENCES

[1] iFlow Family of Processors, Silicon Access. http://www.siliconaccess.com.

[2] Intel IXP Family of Network Processors. http://www.intel.com.

[3] S. Albers and H. Koga. Page migration with limited local memory capacity. In *Workshop on Algorithms and Data Structures*, pages 147–158, August 1995.
[4] 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.
[5] 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.
[6] A. Borodin and R. El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, Cambridge, 1998.
[7] P. Brucker. *Scheduling Algorithms*. Springer-Verlag, Berlin, 2001.
[8] 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.
[9] 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.
[10] 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.
[11] 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.
[12] B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. *Journal of the ACM*, 47:617–643, 2000.
[13] R. Kokku. *ShaRE: Run-time System for High-performance Virtualized Routers*. PhD thesis, Department of Computer Science, University of Texas at Austin, August 2005.
[14] M. S. Manasse, L. A. McGeoch, and D. D. Sleator. Competitive algorithms for server problems. *Journal of Algorithms*, 11:208–230, 1990.
[15] C. A. Phillips, C. Stein, E. Torng, and J. Wein. Optimal time-critical scheduling via resource augmentation. *Algorithmica*, pages 163–200, 2002.
[16] S. S. Seiden. More multiprocessor scheduling with rejection. Technical Report Woe–16, TU Graz, 1997.
[17] S. S. Seiden. *Randomization in On-line Computation*. PhD thesis, University of California, Irvine, 1997.
[18] D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28:202–208, 1985.
[19] 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.
[20] 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.
[21] J. S. Turner. New directions in communications (or which way to the information age?). *IEEE Communications Magazine*, 24(10):8–15, October 1986.
[22] 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.
[23] N. E. Young. On-line file caching. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 82–86, January 1998.

APPENDIX

A. PREFIX PAGING

In this section, we define and solve a variant of the traditional disk paging problem, which we refer to as prefix paging. Let m denote the cache size given to the offline algorithm. In the prefix paging problem, 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 input to the prefix paging problem is a sequence of page requests. This sequence 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.

A.1 Algorithm Mark

Given an arbitrary input to the prefix paging problem, we partition the input into epochs as follows. If fewer than $2m$ distinct pages are accessed in the input, then there is just one epoch. Otherwise, the first epoch is the shortest prefix p of the input 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 input.

Our online algorithm, 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 [6, 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.

A.2 Analysis

LEMMA A.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. □

The following corollary is used in Appendix B.

COROLLARY A.1. *Immediately after processing a given segment, the cache contains all of the pages accessed during the segment.*

PROOF. This is immediate from Lemma A.1, since each epoch consists of a whole number of segments. □

LEMMA A.2. *Algorithm Mark is resource competitive for the prefix paging problem.*

PROOF. By Lemma A.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 offline algorithm has a cache size of m, it 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 input with at least one complete epoch. It remains to consider inputs consisting of a single incomplete epoch. Fix such an input 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. □

B. A RATE-LIMITED PROBLEM

In this section, we solve the special case of $[\Delta \mid d_\ell \mid 1 \mid 1]$ inn which at most m jobs arrive per round, where m is the number of resources used by any offline algorithms. We refer to this special case as rate-limited $[\Delta \mid d_\ell \mid 1 \mid 1]$. Our algorithm for this special case is invoked by algorithm Split in Section 6.2.2 in the case where $\Delta < d_\ell$, for all ℓ. The more challenging case, where $\Delta \geq d_\ell$, for all ℓ, is addressed in Section 5.

B.1 Algorithm RL-Serialize

Our online algorithm, which we refer to as algorithm RL-Serialize, uses $3m$ resources and proceeds in the following three steps. First, given any instance I for unit delay with restricted input, we construct an instance I' for prefix paging as follows. The cache size associated with I' is equals the number of resources associated with I. Let σ be the input sequence associated with I. For any nonnegative integer i, let σ_i be request i of σ. Let $X_i = \cup_\ell \{(\ell, j) \mid 0 \leq j < q_{i,\ell}\}$, where $q_{i,\ell}$ is the number of color ℓ jobs in σ_i. Let σ'_i be a sequence of requests obtained by ordering the files in X_i arbitrarily. The input sequence σ' associated with I' is obtained by concatenating the σ'_i's in increasing order of i.

Second, we use algorithm Mark of Appendix A on σ' to generate a solution S' for I'.

Third, we construct a solution S for I from S' as follows. Fix an arbitrary nonnegative integer i. In the following we describe the schedule of S in round i. Consider each nonnegative integer k such that $0 \leq k < 3m$. Let (ℓ, j) be the page cached at location k immediately after serving the last request in σ'_i in S'. In S, we configure resource k with color ℓ in round i. In round i, S executes as many jobs as its new configuration allows.

B.2 Analysis

We define a schedule S for unit delay with restricted input to be *drop-free* if S does not incur any drop cost.

LEMMA B.1. *For any offline solution T for I, there exists a drop-free solution T' for I such that the cost incurred by T' is at most twice that incurred by T.*

PROOF. We construct T' from T round by round. The schedule of T' for an arbitrary round i is constructed in the following two steps. First, for each job x executed by T in round i, we schedule x on resource k in round i, where resource k is the resource on which x is executed in round i in T. Second, while there exists a job x dropped by T in round i, we schedule the job on an arbitrary idle resource, that is, a resource on which no job is scheduled in round i; note that such a resource can always be found since at most m jobs arrive per round.

The schedule T' does not incur any drop cost. For any color ℓ *job* dropped by T, T' reduces the drop cost by d_ℓ, and increases the reconfiguration cost by at most 2Δ. Since, $\Delta < d_\ell$, for all ℓ, the cost incurred by T' is at most twice that incurred by T. Hence, the lemma follows. □

We omit the proof for Lemma B.2 because it is analogous to the proof for Lemma 5.1 and in fact simpler.

LEMMA B.2. *For any drop-free offline solution T for I with cost C and m resources, there exists an offline solution T' for I' with cache size m and that makes at most $\frac{2C}{\Delta}$ misses.*

LEMMA B.3. *The cost incurred by S is at most Δ times the number of misses incurred by S'.*

PROOF. It is not hard to see that the reconfiguration cost incurred by S is at most Δ times the number of misses incurred by S'. It remains to show that the S does not incur any drop cost.

Fix an arbitrary round i and color ℓ. Let $k_{i,\ell}$ denote the number of resources configured with color ℓ in round i in S. Let $p_{i,\ell}$ denote the number of color ℓ pages in the cache immediately after processing σ'_i in S'. Let $q_{i,\ell}$ denote the number of color ℓ pages in σ'_i. By the construction of S from S', $k_{i,\ell} = p_{i,\ell}$. By Corollary A.1, pages in σ'_i are cached immediately after processing σ'_i. Hence, $p_{i,\ell} = q_{i,\ell}$. By the construction of σ'_i, $q_{i,\ell}$ equals the number of color ℓ jobs in σ_i. Therefore, $k_{i,\ell}$ equals the the number of color ℓ jobs in σ_i, that is, all jobs of color ℓ in σ_i are executed in S. □

THEOREM 5. *Algorithm RL-Serialize is resource competitive for unit delay with restricted input.*

PROOF. Consider any offline solution T for I with cost C using m resources. By Lemma B.1, there exists a drop-free offline solution for I with cost at most $2C$ using m resources. By Lemma B.2, there exists an offline solution for I' with at most $\frac{4C}{\Delta}$ misses using m resources. By Lemma A.2, S' incurs $O(\frac{C}{\Delta})$ cost using $3m$ resources. By Lemma B.3, the schedule S, that is, the solution generated by RL-Serialize, incurs $O(C)$ cost using $3m$ resources. □