Multicast Pull Scheduling: When Fairness is Fine

Jeff Edmonds *   Kirk Pruhs †

April 29, 2002

Abstract

We investigate server scheduling policies to minimize average user perceived latency in pull-based client-server systems (systems where multiple clients request data from a server) where the server answers requests on a multicast/broadcast channel. We first show that there is no $O(1)$-competitive algorithm for this problem. We then give a method to convert any nonclairvoyant unicast scheduling algorithm A to nonclairvoyant multicast scheduling algorithm B. We show that if A works well works well, when jobs can have parallel and sequential phases, then B works well if it is given twice the resources. More formally, if A is an s-speed c-approximation unicast algorithm, then its counter part algorithm B is a $2s$-speed c-approximation multicast algorithm. In is already known [6] that Equi-partition, which devotes an equal amount of processing power to each job, is an $(2+\epsilon)$-speed $O(1+1/\epsilon)$-approximation algorithm for unicast scheduling of such jobs. Hence, it follows that the algorithm BEQUI, which broadcasts all requested files at a rate proportional to the number of outstanding requests for that file, is an $(4+\epsilon)$-speed $O(1+1/\epsilon)$-approximation algorithm. We give another algorithm BEQUI-EDF, and show that BEQUI-EDF is also an $(4+\epsilon)$-speed $O(1+1/\epsilon)$-approximation algorithm. However, BEQUI-EDF has the advantage that the maximum number of preemptions is linear in the number of requests, and the advantage that no preemptions occur if the data items have unit size.

1 Introduction

We investigate server scheduling policies to minimize average user perceived latency in pull-based client-server systems (systems where multiple clients request data from a server) where the server answers requests on a broadcast channel. One notable commercial example of such a system is Hughes' DirecPC system [4]. In the DirecPC system the clients request web files via a low bandwidth dial-up connection, and the web files are broadcasted via high bandwidth satellite to all clients (so it may be possible to satisfy many requests to a common file with a single broadcast). One would expect that the ability to broadcast would reduce the workload on a server for the same reason that proxy caches reduce the workload on a server, because it is common for different clients to make requests for the same data item. The average user perceived latency, or equivalently average flow time, is the average (over all client requests) of the difference in time between when the request is fully satisfied and when the request was made.

For convenience, we will adopt terminology appropriate for the DirecPC system, i.e. a web server broadcasting files to clients, although our discussion is independent of the type of server, and the type of data items. We consider what appear to us to be the two most natural job environments. In the job environment $r_i, pmtn$ the files sizes are not uniform, and the server may preempt, that is, terminate the broadcast of one file, and later return to broadcasting that file from the point of preemption. Systems that service jobs with widely disparate

*York University, Canada. jeff@cs.yorku.ca. Supported in part by NSERC Canada.

†Computer Science Department. University of Pittsburgh. kirk@cs.pitt.edu. Supported in part by NSF grants CCR-9734927, CCR-0098752, ANIR-0123705, and by a grant from the US Air Force.

resource requirements, e.g. an operating system, generally need to allow preemption to achieve reasonable system performance. This would be an appropriate job environment for a web server. In the case of a name server communicating IP address, or any server where all data items are small, a more appropriate job environment would be $r_i, p_i = 1$, that is, jobs are of unit size and preemption is not allowed. Implementable algorithms must be online, that is, the schedule must be created over time as the jobs arrive without knowledge of the future. We make the natural assumption that servers broadcast files sequentially, and the clients must receive files sequentially, that is, a client cannot buffer the last part of a file if it makes a request for that file in mid-broadcast of that file. Although recent results in [12] on the relationship of various models of multicast pull scheduling show that the results in this paper extend, with at most minor modification, to other models (e.g. when the client can buffer the end of a file). We will denote the two resulting problems as $B|r_i, pmtn|\sum F_i$ and $B|r_i, p_i = 1|\sum F_i$.

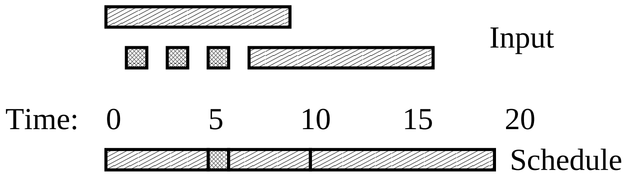

Figure 1: An example instance of $B|r_i, pmtn|\sum F_i$

As a concrete example of the problem $B|r_i, pmtn|\sum F_i$ consider the instance shown in figure 1. There are two files (designated by the rectangles with two different fill patterns in the input). The first file is of length 9, and the second file is of length 1 (designated by the horizontal lengths of the rectangles in the input). The first file is requested at times 0 and time 7 (designated by the horizontal positioning of the rectangles in the input). The second file is requested at times 1, 3 and 5. Presumably the online scheduler, with unit bandwidth, starts broadcasting the first file at time 0. At time 1, a request for the second file arrives and the online scheduler must decide whether to continue broadcasting the first file, or preempt and start broadcasting the second file. In the feasible schedule shown, the second file is broadcast from time 5 to time 6, satisfying all of the requests for the second file with this single broadcast. Note that after this schedule finished broadcasting the first file at time 10, the schedule must again broadcast all 9 units of the first file since we are assuming that the user that made the request for the first file at time 7 could not buffer the last 3 units of the first file between time 7 and time 10. The average flow time for this schedule is $(10 + 5 + 3 + 1 + 12)/5$, where the flow times of the individual requests in the numerator are ordered by increasing the arrival times of the requests.

In section 3 we will show that there is no $O(1)$-competitive algorithm for the multicast pull scheduling problem $B|r_i, pmtn|\sum F_i$. In [9] it was shown that there are no $O(1)$-competitive algorithms for the problem $B|r_i, p_i = 1|\sum F_i$. Note that the lower bound for $B|r_i, p_i = 1|\sum F_i$ does not apply to the the problem $B|r_i, pmtn|\sum F_i$ because preemption is allowed in $B|r_i, pmtn|\sum F_i$.

We thus consider resource augmentation analysis. Resource augmentation analysis was proposed as an method for analyzing scheduling algorithms in [8]. We adopt the notation and terminology from [11]. In our context of a scheduling minimization problem, an *s-speed c-approximation* algorithm A has the property that for all inputs, the value of the objective function of the schedule that A produces with speed s processors, denoted by A_s, is at most c times the optimal value of the objective function for speed 1 processors. Standard competitive analysis assumes that A has unit speed processors. So an algorithm A is *c-competitive* if A is a 1-speed c-approximation algorithm. Not that an s-speed c-approximation algorithm is $O(c)$-competitive if $\text{OPT}_1 = \Theta(\text{OPT}_s)$. Intuitively, OPT_1 is $\Theta(\text{OPT}_s)$ unless the system has load at least $1/s$. So one way to interpret an s-speed c-approximation analysis result is that the algorithm should perform reasonably well if the system is not too heavily loaded. The only previously known positive result for broadcast scheduling was an $O(1)$-speed $O(1)$-approximation polynomial-time LP-based *offline* algorithm given in [9] for the problem $B|r_i, p_i = 1|\sum F_i$. The constants were subsequently improved in [5, 7].

Broadcasting complicates the task of the server in that it must decide whether to broadcast a file as soon as possible after a request comes or to wait for some undetermined length of time in case more requests for the same file arrive that can be handled by a single broadcast. To further understand this complication, let us consider the lower bound proofs that no $O(1)$-competitive online algorithms exist for $B|r_i, p_i = 1|\sum F_i$ or $B|r_i, pmtn|\sum F_i$. After the online algorithm has performed a significant amount of work on a file that was requested by a single client, the adversary can again direct another client to request that file. The online algorithm must service this second request as well. In contrast, the optimal schedule knows not to initially give any resources to the first request, because the broadcast for the second request simultaneously services the first. In this regard, the work associated with the first request is essentially "sequential", in that even though the online algorithm devotes a lot of resources to it and the optimal algorithm devotes no resources to it, it completes under both within a constant factor of the time. This is in contrast to the more standard notion of *parallel* work, where increasing the fraction of the processing power devoted to this work by a multiplicative factor of f decreases the time required to complete the work by a factor of f. Hence, we interpret these lower bounds to indicate that the real difficulty of broadcast scheduling is that the adversary can force some of the work to be sequential. In section 4, we formalize the above intuition by giving a method to convert any nonclairvoyant unicast scheduling algorithm A to nonclairvoyant multicast scheduling algorithm B. We show that if A works well works well, when jobs can have parallel and sequential phases, then B works well if it is given twice the resources. The basic idea is that B simulates A, creating a separate job for each request, and then the amount of time that B broadcasts a file is equal to the amount of time that A runs the corresponding jobs. More formally, if A is an s-speed c-approximation unicast algorithm, then its counter part algorithm B is a $2s$-speed c-approximation multicast algorithm. This result is surprising given the fact the problems being solved are so different.

It is shown in [6] that an algorithm, Equi-partition, which devotes an equal amount of processing power to each job, is an $(2+\epsilon)$-speed $O(1+1/\epsilon)$-approximation algorithm for unicast scheduling of jobs with "natural" speed-up curves. These "natural" speed-up curves include jobs that consist of parallel and sequential phases. In section 5 we apply our reduction to Equi-partition to obtain the intuitive algorithm BEQUI for the problem $B|r_i, pmtn|\sum F_i$. The algorithm BEQUI broadcasts each file at a rate proportional to the number of outstanding requests for that file. Hence we can conclude that BEQUI is an $(4+\epsilon)$-speed $O(1+1/\epsilon)$-approximation algorithm for the problem $B|r_i, pmtn|\sum F_i$. We believe that these results are evidence of the applicability to other problems of the model of scheduling jobs with various speed-up curves, and the analysis of Equi-partition from [6].

In section 6, we propose another algorithm BEQUI-EDF. We show that BEQUI-EDF is also an $(4+\epsilon)$-speed $O(1+1/\epsilon)$-approximation algorithm for $B|r_i, pmtn|\sum F_i$. However, BEQUI-EDF has the advantage of guaranteeing that it preempts on average at most once per broadcast, or alternatively, that the number of preemptions is at most the number of requests. Furthermore, we show that if the files have unit size then BEQUI-EDF will not preempt any broadcast. Thus BEQUI-EDF is an $(4+\epsilon)$-speed $O(1+1/\epsilon)$-approximation algorithm for $B|r_i, p_i = 1|\sum F_i$. The algorithm BEQUI-EDF can be viewed as a two stage algorithm. The first stage BEQUI-EDF simulates the algorithm BEQUI. When BEQUI completes the jth broadcast of a file at time t, a job with work equal to the length of this file, and deadline equal to t plus the flow time for the jth broadcast in BEQUI, is sent to the second stage. The second stage is always broadcasting the job with the earliest deadline.

The algorithms BEQUI and BEQUI-EDF have several nice features. Both algorithms are nonclairvoyant, meaning that they do not need to know the quantity of remaining unfinished work for each uncompleted job. This could allow a server to begin broadcasting a dynamically generated file before the file is fully generated. In addition, both algorithms are fair to all jobs, and thus they avoid starving long jobs as say Shortest Remaining Processing Time might. Both algorithms are easy to implement. Another way of viewing BEQUI is that the bandwidth is distributed evenly between all requests $R_{i,j}$ that are either waiting or being serviced. Hence, BEQUI can be implement by servicing the requests in a round robin fashion.

Multicast pull scheduling has received considerable attention recently in the database, networks, and algorithms literature (e.g. [1, 3, 9]). Let us quickly summarize a couple of the seemingly most related results in the literature that we have not yet mentioned. The offline problem is NP-hard [5]. In [3] it is shown that the algorithm First-Come-First-Served is 2-competitive for the objective function of minimizing the max flow time, and that minimizing the maximum flow time can be approximated well offline (under the assumption that client buffering is allowed). There has been a fair amount of research into push-based broadcast systems, sometimes called broadcast disks, where the server pushes information to the clients without any concept of a request, ala a television or radio broadcast [2, 10]. It seems that good strategies for push-based systems have little to do with good strategies for pull-based systems.

Before preceding with the results we need to formalize some terms and concepts in section 2.

2 Definitions

We start by more formally defining the multicast pull scheduling problem $B|r_i, pmtn|\sum F_i$. The input that the server receives is a sequence of requests for m different files. The j^{th} request for the i^{th} file is denoted $R_{i,j}$ and its integer arrival time is denoted by $a_{i,j}$. An online server algorithm does not become aware of $R_{i,j}$ until time $a_{i,j}$. The length/work of file i is an integer l_i. The only information that a nonclairvoyant algorithm can infer about l_i is that it is lower bounded by the amount of work that the algorithm has done on the file. Let R denote the collection of all requests.

A multicast scheduling algorithm, here denoted B_s, has a given speed s. Broadcasting a file simultaneously *services* all requests that arrived before the broadcast began. With this goal in mind, the algorithm must schedule when to broadcast the files at speed s across the single channel of bandwidth L. At each time step, the algorithm partitions the effective sL bandwidth between the files. We denote by $p^B_{i,t}$ the bandwidth allocated by $B_s(R)$ to the i^{th} file at time t. The restriction is that $\sum_i p^B_{i,t} \leq sL$. We denote by $b^{B,k}_i$ the time that the k^{th} broadcasting of the i^{th} file begins (that is the time that the first bit of i^{th} file is broadcast for the k^{th} time), and by $c^{B,k}_i$ the time that the k^{th} broadcasting of the i^{th} file ends. To accomplish this, the algorithm must allocate enough bandwidth so that $\int_{t\in[b^{B,k}_i, c^{B,k}_i]} p_{i,t} = l_i$, because this is the length of the file.

The life span of a request $R_{i,j}$ is as follows. It *arrives* at some time $a_{i,j}$ specified by the input R. It must *wait* until the next broadcasting of the i^{th} file to begin, i.e. the minimum k such that $b^{B,k}_i \geq a_{i,j}$. This time is denoted $b^B_{i,j} = b^{B,k}_i$. The request is then being *serviced* until the time denoted $c^B_{i,j} = c^{B,k}_i$ that this broadcast completes. The flow time for this request is time from the arrival and the completion the request, namely $[c^B_{i,j} - a_{i,j}]$. The *total/average flow time* of a schedule is the total/average of this over all requests. We use BOPT to denote the optimal schedule. Note for simplicity we polymorphically use the same notation $B_s(R)$ for the algorithm B_s on input R, the schedule produced by B_s on R, and the total flow time for this schedule.

The job environment $SC, r_i, pmtn$ given in [6] is as follows. The input to the problem is a set of n jobs $J = \{J_j\}$ that are to be executed on P processors (note that generally speaking, the number of processors is not relevant since one can use the speed-up curves to simulate a multi-processor environment in single machine environment). The j^{th} job is denoted by J_j, its arrival time is denoted by a_j, and it has a sequence of phases $\left\langle J_{j,1}, J_{j,2}, \ldots, J_{j,q_j}\right\rangle$. Each phase is an ordered pair $\langle w_{j,q}, \Gamma_{j,q}\rangle$, where $w_{j,q}$ denotes the amount of *work* and $\Gamma_{j,q}$ denotes its *speedup function*. Here $\Gamma_{j,q}(\beta)$ represents the rate at which work is executed for phase q of job i when given β processors (note that β need not be integer). The original motivation for introducing speed-up curves was that, in the context of a parallel computer, not all code is equally parallelizable. In this paper, each phase can either be a *parallel* phase, that is a phase where $\Gamma(\beta) = \beta$, or *sequential* phase, that is a phase where $\Gamma(\beta) = 1$. (Note that in [6] each speedup function can be an arbitrary nondecreasing sublinear function.) Sequential work completes work at a rate of 1 even when absolutely no processors are allocated to it. A unicast scheduling algorithm A_s has a given speed s. At each time step, the algorithm partitions the effective sP processors between the jobs. We denote by $p^A_{j,t}$ the number of processors allocated by $A_s(J)$ to the job J_j at time t. The restriction is that $\sum_j p^A_{j,t} \leq sP$. We denote by $c^A_{j,q}$ the time that $A_s(J)$ completes the q^{th} phase of job J_j. To accomplish this, the algorithm must allocate enough processors so that $\int_{t\in[c^A_{j,q-1},c^A_{j,q}]} \Gamma_{j,q}(p^A_{j,t})dt = w_{j,q}$, because $w_{j,q}$ is the work in this phase and $\Gamma_{j,q}$ is its speedup function. The *completion* time of a job J_j, denoted c^A_j, is the completion time of the last phase of the job. The flow time for this job is $[c^A_j - a_j]$. The performance of the algorithm is measured by the average flow time, $A_s(J) = \text{Avg}_j[c^A_j - a_j]$. We use JOPT to denote the optimal schedule.

In this setting a *nonclairvoyant* schedule is completely in the dark. In addition to not knowing what jobs will arrive in the future, it does not know the amount of work remaining or the speedup functions $\Gamma_{j,q}$ of the jobs that have already arrived. All it knows is when a job arrives and when it completes. Not knowing the amount of work remaining in a job, prevents the scheduler from doing Shortest-Remaining-Work-First, which is optimum when all the jobs are fully parallelizable. Not knowing the speedup functions of the jobs is an even bigger handicap because any processors accidently allocated to a sequential phase of a job are effectively wasted. The optimum scheduler, in contrast, knows not to allocate any processors to the sequential phases of jobs. A nonclairvoyant scheduler that is often used in practice for jobs with varying speed-up curves is Equi-partition, which allocates an equal number of processors to each outstanding job. That is, $p^E_{j,t} = sP\frac{1}{n^E_t}$, where n^E_t is the number of jobs that are outstanding at time t, and for simplicity we use E to denote the algorithm Equi-partition. In [6], it is shown that Equi-partition is a $2+\epsilon$-speed $O(1+\frac{1}{\epsilon})$-approximation algorithm (note that is it easy to see that no $O(1)$-competitive algorithm exists for this problem).

Theorem 1 ([6]) For all instance J of $SC, r_i, pmtn$, $\frac{E_{2+\epsilon}(J)}{\text{JOPT}_1(J)} = O(1+\frac{1}{\epsilon})$.

3 A Lower Bound

Theorem 2 There is no $o(\sqrt{n})$-competitive algorithm for $B|r_i, pmtn|\sum F_i$, where n is the number of requests.

Proof: We prove the first statement first. Initially $2k$ unit-length files are each requested once. Without loss of generality, assume that at time k that the online algorithm has done less work on the first k files than on the last k files. Thus the online algorithm has at least $k/2$ work remaining on the first k files just before time k. At time k, the last k files are all requested again by one more client each. Thus at time k, the online has $3k/2$ remaining work. At time $2k$, the online algorithm must have at least $k/2$ remaining work. The adversary can have no remaining work at time $2k$ by broadcasting the first k files from time 0 to k, and by broadcasting the last k files between time k and time $2k$. ¿From time $2k$ until time k^2 a new previously unrequested unit length file is requested at each integer time step. Thus at each time between time $2k$ and time k^2, the online algorithm with have at least $k/2$ unfinished work, and hence at least $k/2$ unsatisfied requests. Therefore the total flow time of the online algorithm is $\Omega(k^3)$. The optimal total flow time is $O(k^2)$. The number of requests $n = \Theta(k^2)$. ■

4 A Reduction from Multicast to Unicast

In this section we give a reduction that converts a nonclairvoyant unicast scheduling algorithm into an almost equally competitive nonclairvoyant multicast scheduling algorithm.

Description of the Reduction: Let A be a nonclairvoyant algorithm for the unicast scheduling problem $P|SC, r_i, pmtn|\sum F_i$. Despite the fact that the problems being solved are so different, our corresponding algorithm

B for the multicast pull scheduling problem $B|r_i, pmtn|\sum F_i$ requires little changes. Algorithm B is defined as follows. Algorithm B broadcasts each of its files continuously, (though sometimes at a rate of zero). When it completes its k^{th} broadcast of the i^{th} file, it immediately starts its $(k+1)^{st}$ broadcast of the file, i.e. $c_i^{B,k} = b^{B,k+1}$. Algorithm B simulates algorithm A to determine the current rate at which it broadcasts each file. At each point in time, the jobs active under A directly correspond to the requests that are either waiting to be serviced, or are currently being serviced under B. When B receives the j^{th} request for the i^{th} file, $R_{i,j}$, B tells A that it has received a new job $J_{(i,j)}$. When B competes servicing a request, it tells A that the corresponding job has completed. Because algorithm A is nonclairvoyant, it never has any more knowledge about the jobs then this. Hence, B can know at each point in time the number of processors $p^A_{(i,j),t}$ allocated by A to job $J_{(i,j)}$. Algorithm B then allocates the same amount of resources to each request by broadcasting the i^{th} page at a rate of $p^B_{i,t} = \sum_j p^A_{(i,j),t}$. Note that the total bandwidth needed by B is always equal to the number sP of processors that A has.

Theorem 3 If A is an s-speed c-approximation nonclairvoyant algorithm for the problem $P|SC, r_i, pmtn|\sum F_i$ then B is an $2s$-speed c-approximation nonclairvoyant algorithm for the problem $1|SC, r_i, pmtn|\sum F_i$.

Proof: We start by making one simplification. We assume that the $(k+1)^{st}$ broadcast for B begins when the k^{th} broadcast ends, that is, $c_i^{B,k} = b^{B,k+1}$. Thus we assume that any requests that arrive strictly after $c_i^{B,k}$ will not be serviced by the $(k+1)^{st}$ broadcast for B. This assumption is strictly to B's disadvantage. The assumption that $c_i^{B,k} = b^{B,k+1}$ allows us to simplify notation and our figures.

Changing s, we will assume that A is $\frac{s}{2}$ competitive and prove that B is s competitive. To do this we must prove that for each set of requests R, that the flow time of B with requests R and speed s (denoted $B_s(R)$) is at most a constant times the flow time of the optimal schedule $\mathrm{BOPT}_1(R)$ for the requests with speed 1. This is done by constructing from R a set of jobs J to give to the algorithm A and proving that $\frac{B_s(R)}{\mathrm{BOPT}_1(R)} \le \frac{A_s(J)}{\mathrm{JOPT}_2(J)} \le O(1)$. The last inequality follows from the fact that algorithm A is $\frac{s}{2}$ competitive.

The first step of the proof considers in detail the execution of $B_s(R)$. The second step quickly considers the execution of the optimal algorithm $\mathrm{BOPT}_1(R)$. The third step uses the details of these two executions to construct the sequential and parallel phases of the jobs in J in such a way that the flow time for each request in $B_s(R)$ will be equal to (or be less than) the flow time of the corresponding job in $A_s(J)$. The final step constructs a schedule $\mathrm{JOPT}'_2(J)$ whose total flow time is at most the total flow time of $\mathrm{BOPT}_1(R)$. The optimal total flow time of $\mathrm{JOPT}_2(J)$ is clearly at most the total flow time of $\mathrm{JOPT}'_2(J)$. These steps complete the proof.

The proof requires lots of notation. It considers five different algorithms, $B_s(R)$, $\mathrm{BOPT}_1(R)$, $A_s(J)$, $\mathrm{JOPT}'_2(J)$ and $\mathrm{JOPT}_2(J)$. For simplicity, the first four will be indicated by a superscript of B, O, A, and O'. Because the proof focuses exclusively on the i^{th} file, we will drop the subscript i. For example, R_j and J_j, and not $R_{i,j}$ and $J_{(i,j)}$, will denote the j^{th} request of the i^{th} file and the job related to it, respectively. We use a_j, b_j^B, and c_j^B to denote the times that request R_j for the i^{th} file arrives, begins, and completes being broadcasted by $B_s(R)$. We use $b^{B,k}$ and $c^{B,k}$ to denote the times that the k^{th} broadcast of the i^{th} file begins and completes in $B_s(R)$. We use p_t^B to denote the bandwidth allocated to the i^{th} file at time t by B, and similarly we use $p^A_{j,t}$ to denote the number of processors that A devotes to job $J_{(i,j)}$ at time t. Finally, we use $j^{B,k}$ to be the index of the first request that arrives after the beginning time $b^{B,k}$ of the k^{th} broadcast, i.e. the smallest index j such that $a_j^B > b^{B,k}$.

In the first step of the proof we attempt to understand the execution of $B_s(R)$. Figure 2 depicts the execution for the i^{th} file by $B_s(R)$ on requests R. The curve increasing up and to the right graphs the arrival time a_j of the j^{th} request R_j for the i^{th} file. Note that time t is on the vertical axis and the index j of request for this file is on the horizontal axis. The horizontal dotted lines depict the times at which a broadcast of the i^{th} file begins, and the horizontal dashed lines depict the times at which a broadcast of the i^{th} file ends. Imagine any vertical line through Figure 2 at some index j. The life of a request can be determined by tracing this vertical line from

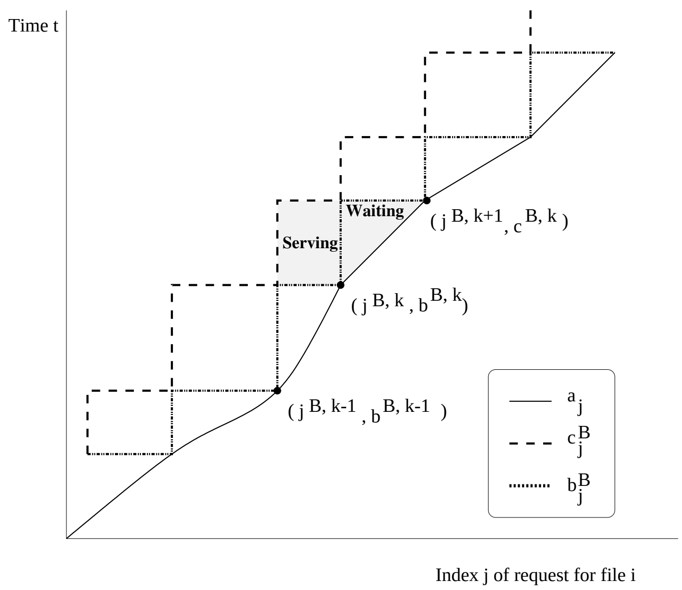

Figure 2: The computation of $B_s(R)$.

bottom to top. The request arrives at the time a_j indicated by the solid curve. It waits until time b_j^B, indicated by the dotted line that intersects the vertical line, at which time it begins to receive the i^{th} file from the next broadcast. It is served until the time denoted c_j^B, indicated by the dashed line intersecting the vertical line, at which time this broadcast completes. The flow time for the request is then $c_j^B - a_j$.

Now let us focus on the k^{th} broadcast of the i^{th} file, and consider Figure 2 again. The top two dots depict that this broadcast begins at the time denoted $b^{B,k}$ and completes at the time denoted $c^{B,k}$. They also depict that the range of requests R_j that arrive during this broadcast. Hence, during this broadcast, these requests are waiting to be serviced. They will be serviced later by the $k+1^{st}$ broadcast. The bottom two dots in Figure 2 depict that the range of requests R_j that that are serviced by this k^{th} broadcast.

The total number of packets broadcasted by the k^{th} broadcast of the i^{th} file of size l_i is computed to be

$$\begin{aligned}
l_i &= \int_{t \in [b^{B,k}, c^{B,k}]} p_t^B \, t \\
&= \sum_{j \in \{j \mid R_j \text{ is waiting or being serviced at time } t\}} p_{j,t}^A \\
&= \text{the volume under the highlighted step in Figure 1 if you assume that point } \langle j,t\rangle \text{ has height } p_t^E. \\
&= \sum_{j \in (j^{B,k-1}, j^{B,k}]} \int_{t \in [b^{B,k}, c^{B,k}]} p_{j,t}^A \, t + \sum_{j \in (j^{B,k}, j^{B,k+1}]} \int_{t \in [a_j, c^{B,k}]} p_{j,t}^A \, t
\end{aligned}$$

The second step of the proof considers the execution of $\mathrm{BOPT}_1(R)$. The k^{th} broadcast under $\mathrm{BOPT}_1(R)$ of the i^{th} file is denoted to begin at time $b^{O,k}$ and complete at time $c^{O,k}$. We do not assume that the file is broadcasted continually, i.e. that $c^{O,k} = b^{O,k+1}$. In fact, we do not even assume that the broadcasts to this file are disjoint, i.e. that $c^{O,k} \leq b^{O,k+1}$. $\mathrm{BOPT}_1(R)$ dedicates a total of l_i its 1 speed bandwidth during the time interval $[b^{O,k}, c^{O,k}]$ to complete this k^{th} broadcast of the i^{th} file, and because of this broadcast, the requests R_j that arrive during the time interval $(b^{O,k-1}, b^{O,k}]$ complete at time $c_j^O = c^{O,k}$.

The third step of the proof (see Figure 3) uses the details of $B_s(R)$ and $\mathrm{BOPT}_1(R)$ to construct the sequential and parallel phases of the jobs in J in such a way that the flow time for each request in $B_s(R)$ will be equal to (or be less than) the flow time of the corresponding job in $A_s(J)$. Because each job is designed to arrive at the same time as the corresponding request, it is sufficient to prove that the job completes no earlier, i.e. that $c_j^A \geq c_j^B$. In Figure 3 the dashed line plots the completion time c_j^B of the j^{th} request for the i^{th} file under $B_s(R)$. The dotted line plots the times b_j^O that the requests under $\mathrm{BOPT}_1(R)$ begin being served. There are two cases in determining how job J_j is designed depending on how these two lines relate.

In the first case, $B_s(R)$ completes the request R_j before $\mathrm{BOPT}_1(R)$ begins the request, i.e. $c_j^B \leq b_j^O$. In this case, request R_j is replaced with a job J_j that is completely sequential arriving at time a_j with work $c_j^B - a_j^B$ so that it completes at time c_j^B. Recall that sequential work complete at unit rate independent of the number of processors allocated to them.

In the second case, $B_s(R)$ completes the request after $\mathrm{BOPT}_1(R)$ begins the request, i.e. $c_j^B > b_j^O$. In this case, request R_j is replaced with a job J_j that has a sequential phase followed by a parallel phase. The job arrives at time a_j and the sequential phase has work $b_j^O - a_j^B$ so that it completes at time b_j^O. The parallel phase has an infinitesimally more work than the amount $\int_{t \in [b_j^O, c_j^B]} p_{j,t}^A$ completed by the processors allocated to it by $A_s(J)$ during the time period $[b_j^O, c_j^B]$. Hence, at time c_j^B the job will still have an infinitesimally small amount of work left on the job. This extra work is enough to insure that even if $A_s(J)$ is not working on the job at time c_j^B, it completes the job some time after this required time. This completes the third step of the proof.

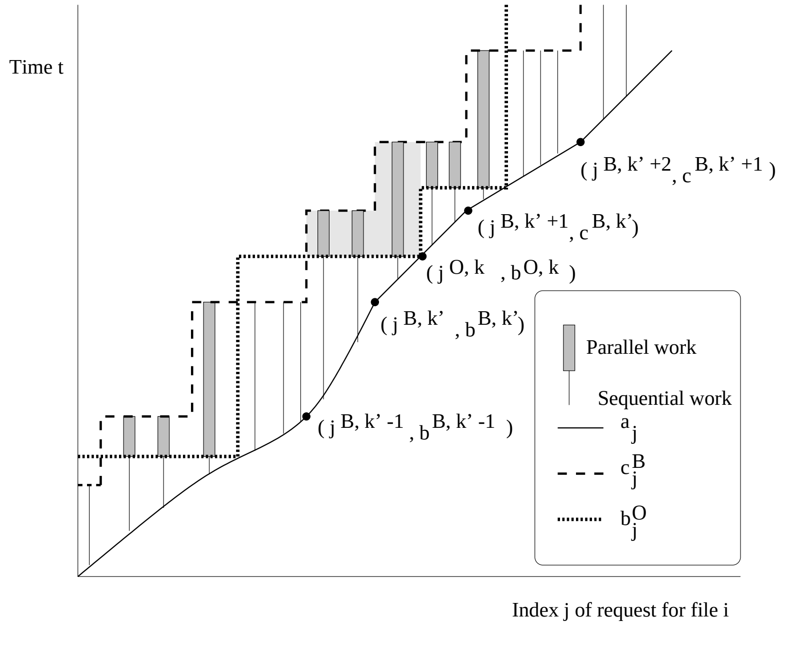

Figure 3: The reduction from Broadcasting to the job model.

The final step of the proof converts the given algorithm $\mathrm{BOPT}_1(R)$ into an algorithm denoted $\mathrm{JOPT}'_2(J)$ that completes our constructed jobs J with a total flow time that is at most that of $\mathrm{BOPT}_1(R)$. This conversion is done one broadcast at a time. Consider the k^{th} broadcast of the i^{th} file by $\mathrm{BOPT}_1(R)$ which begins at some time $b^{O,k}$ and completes at some time $c^{O,k}$. The requests completed by this k^{th} broadcast are partitioned into three cases depending on how the request is serviced by $B_s(R)$. There is a unique broadcast of the i^{th} file by $B_s(R)$ that ends next after the time $b^{O,k}$ that the k^{th} broadcast by $\mathrm{BOPT}_1(R)$ begins. Denote by k' the index of this broadcast of $B_s(R)$. More formally, k' is the index for which $c^{B,k'-1} \leq b^{O,k} < c^{B,k'}$.

The first case of the requests R_j serviced by the k^{th} broadcast by $\mathrm{BOPT}_1(R)$ are those whose arrival time a_j is within the time interval $(b^{O,k-1}, b^{B,k'-1}]$. Note that this interval may be empty if $b^{O,k-1} > b^{B,k'-1}$. The bottom of this range is because for a request to be handled by the k^{th} $\mathrm{BOPT}_1(R)$ broadcast, it must arrive after the beginning $b^{O,k-1}$ of the $k-1^{st}$ broadcast. The top of this range is depicted by the bottom dot in Figure 3. The second case are those requests R_j for which $a_j \in (b^{B,k'-1}, b^{B,k'}]$. The third case are those requests R_j for which $a_j \in (b^{B,k'}, b^{O,k}]$. The top of this range is because for a request to be handled by the k^{th} $\mathrm{BOPT}_1(R)$ broadcast, it must arrive not after the beginning of this broadcast. The separation between the second and third cases, when $b^{B,k'} < b^{O,k}$, is depicted by the dot second from the bottom in Figure 3.

Consider the first case, that is, requests R_j that arrive before time $b^{B,k'-1}$. Hence, they are serviced by either the $k'-1^{st}$ broadcast of $B_s(R)$, or an earlier broadcast of $B_s(R)$. Hence, their completion time c^B_j is no later than $c^{B,k'-1}$, which by the definitions of k and k' is no later than $b^O_j = b^{O,k}$. Recall that when constructing jobs J_j from requests R_j, the requirement of the first case was that $B_s(R)$ completes the request before $\mathrm{BOPT}_1(R)$ begins the request, i.e. $c^B_j \leq b^O_j$. In this case, request R_j is replaced with a job J_j that is completely sequential. The algorithm $\mathrm{JOPT}'_2(J)$ knows to allocate no processors to sequential work. Independent of the number of processors allocated to it, job J_j is designed to complete at time c^B_j, which by assumption is before b^O_j when $\mathrm{BOPT}_1(R)$ begins to serves the request, which is certainly before c^O_j when it completes the request. We can then conclude that $\mathrm{JOPT}'_2(J)$ will complete all such requests J_j at least by the time that $\mathrm{BOPT}_1(R)$ does.

In the second and third cases we consider the requests R_j that arrive after time $b^{B,k'-1}$. Hence, their completion time c^B_j is $c^{B,k'}$ or later, which is strictly later than $b^O_j = b^{O,k}$ by the definition of k and k'. It follows that job J_j is constructed to have a sequential phase followed by a parallel phase. The first sequential phase is constructed to complete at time $b^O_j = b^{O,k}$. Recall that during the preceding time interval $[b^{O,k}, c^{O,k}]$, $\mathrm{BOPT}_1(R)$ dedicated bandwidth at least l_i to the i^{th} file. In the remainder of the proof we wish to establish that the parallel work in the second and third case jobs together do not have more than $2l_i$ work. If we can establish this, it will be then be possible for $\mathrm{JOPT}'_2(J)$ to complete all of these jobs during $[b^{O,k}, c^{O,k}]$ since it has a speed 2 processor. Recall that neither JOPT_2 or $\mathrm{JOPT}'_2(J)$ has to work on the sequential phases of these jobs.

The total parallel work of the second and third case jobs is then the volume under the backwards L shaped shaded region in Figure 3. Recall that we use $j^{B,k}$ to be the index of the first request that arrives after the beginning time $b^{B,k}$ of the k^{th} broadcast in $B_s(R)$. We similarly define we use $j^{O,k}$ to be the index of the first request that arrives after the beginning time $b^{O,k}$ of the k^{th} broadcast in $\mathrm{BOPT}_1(R)$.

Because these requests are handled by the k^{th} broadcast of $\mathrm{BOPT}_1(R)$, we know that $b^O_j = b^{O,k}$. The second case requests R_j, those for $j \in [j^{B,k'-1}, j^{B,k'})$, arrive at $a_j \in (b^{B,k'-1}, b^{B,k'}]$, and hence complete at time $c^B_j = c^{B,k'}$. The third case requests R_j, those for $j \in [j^{B,k'}, j^{O,k})$, arrive at $a_j \in (b^{B,k'}, b^{O,k}] \subseteq (b^{B,k'}, b^{B,k'+1}]$ and hence complete at time $c^B_j = c^{B,k'+1}$. We can conclude that the total parallel work of these case 2 and 3 jobs is

$$\begin{aligned} \lrcorner &= \sum_{j \in [j^{B,k'-1}, j^{B,k'})} \int_{t \in [b^{O,k}, c^{B,k'}]} p^A_{j,t} dt \\ &+ \sum_{j \in [j^{B,k'}, j^{O,k})} \int_{t \in [b^{O,k}, c^{B,k'+1}]} p^A_{j,t} dt \end{aligned}$$

Intuitively, if you could lay Figure 2 on top of Figure 3, you would see that each $\lrcorner$ shaped region in figure 3 lies within the region formed by two consecutive broadcasts of $B_s(R)$ in figure 2, which we denote by $\square$. We know that the volume of the region under any one broadcast of $B_s(R)$ is l_i. Hence, the total parallel work that $\mathrm{JOPT}'_2(J)$ must complete on these case 2 and 3 jobs at most $2 \cdot l_i$. We conclude that $\mathrm{JOPT}'_2(J)$ is able to complete all this parallel work by c^O_j since $\mathrm{BOPT}_1(R)$ completes l_i work by time c^O_j with only a unit speed processor.

More formally, the total work in the k'^{th} and the $k'+1^{st}$ broadcasts of Figure 2 is

$$\square = 2l_i$$

$$
\begin{aligned}
= & \sum_{j \in [j^{B,k'-1}, j^{B,k'})} \int_{t \in [c^{B,k'-1}, c^{B,k'}]} p^A_{j,t} dt \\
& + \sum_{j \in [j^{B,k'}, j^{B,k'+1})} \int_{t \in [a_j, c^{B,k'}]} p^A_{j,t} dt \\
& + \sum_{j \in [j^{B,k'}, j^{B,k'+1})} \int_{t \in [c^{B,k'}, c^{B,k'+1}]} p^A_{j,t} dt \\
& + \sum_{j \in [j^{B,k'+1}, j^{B,k'+2})} \int_{t \in [a_j, c^{B,k'+1}]} p^A_{j,t} dt
\end{aligned}
$$

The jobs in the first summand correspond to requests serviced by the broadcast k' of $B_s(R)$, the jobs in the second summand correspond to requests that arrive during broadcast k' of $B_s(R)$ that must wait this broadcast out, the jobs in the third summand correspond to requests serviced by the broadcast $k'+1$ of $B_s(R)$, and the jobs in the fourth summand correspond to requests that arrive during broadcast $k'+1$ of $B_s(R)$ that must wait this broadcast out.

We see that $\leq$ as follows. For the requests within the range $j \in [j^{B,k'-1}, \min(j^{B,k'}, j^{O,k}))$, the time interval $t \in [b^{O,k}, c^{B,k'}]$ in is a subset of the time interval $t \in [c^{B,k'-1}, c^{B,k'}]$ in , because by the definition of of k' it is the case that $c^{B,k'-1} \leq b^{O,k}$. The range of requests $j \in [j^{B,k'}, j^{O,k})$ in is a subrange of the range $j \in [j^{B,k'}, j^{B,k'+1})$ appearing twice in , because $b^{O,k} \leq c^{B,k'}$ by the definition of k and k', and obviously $c^{B,k'} \leq b^{B,k'+1}$. Hence $j^{O,k} \leq j^{B,k'+1}$. Then for these requests, the time interval $t \in [b^{O,k}, c^{B,k'+1}]$ in is a subset of the time interval $t \in [a_j, c^{B,k'}] \cup [c^{B,k'}, c^{B,k'+1}] = [a_j, c^{B,k'+1}]$ in , because $a_j \leq b^{O,k}$ for requests R_j served by $\mathrm{BOPT}_2(R)$'s k^{th} broadcast. Because the sums and integrals are over a subranges, it follows that $\leq$.

To conclude, given a set of requests R, we constructed a set of jobs J in such a way that the flow times $B_s(R) = A_s(J)$ and $\mathrm{BOPT}_1(R) \geq \mathrm{JOPT}'_2(J) \geq \mathrm{JOPT}_2(J)$. This completes the proof. ■

5 BEQUI

We now apply our reduction from section 4 to the Equi-Partition algorithm to construct the algorithm BEQUI. We then obtain the result that BEQUI is a $4+\epsilon$-speed $O(1+\frac{1}{\epsilon})$-approximation algorithm for $B|r_i, pmtn|\sum F_i$.

Description of BEQUI: Let s be the speed of the processor and L the bandwidth of the channel. The algorithm BEQUI broadcasts each file at a rate proportional to the number of outstanding requests. That is, the bandwidth allocated to the i^{th} file at time t is $sL\frac{n^E_{i,t}}{n^E_t}$, where $n^E_{i,t}$ denotes the number of requests for the i^{th} file that are either waiting or being served at time t and $n^E_t = \sum_i n^E_{i,t}$ denotes this number over all files. Note that having either lots of requests waiting for a file or lots of requests being served motivates BEQUI to broadcast the file faster.

Theorem 4 For all sets of requests R, $\frac{\mathrm{BEQUI}_{4+\epsilon}(R)}{\mathrm{BOPT}_1(R)} = O(1+\frac{1}{\epsilon})$.

Proof: This follows by combining theorem 1 and theorem 3. ■

6 BEQUI-EDF

Part of the code of $\mathrm{BEQUI\text{-}EDF}_{(1+\epsilon)(4+\epsilon)}$ simulates the algorithm $\mathrm{BEQUI}_{(4+\epsilon)}$ with a factor $1+\epsilon$ slower bandwidth. We simplify our argument by scaling the broadcasting speed units so that BEQUI has a unit speed processor, and BEQUI-EDF has a $(1+\epsilon)$ speed processor.

Description of BEQUI-EDF$_{1+\epsilon}$: At time $c_i^{E,k}$ when BEQUI$_1$ completes broadcasting the i^{th} file for the k^{th} time, a job $J_{i,k}^{EDF}$ is given to the algorithm EDF$_{1+\epsilon}$ with arrival time $a_{i,k}^{EDF} = \lceil c_i^{E,k} \rceil$, work $l_{i,k}^{EDF} = l_i$, and deadline $d_{i,k}^{EDF} = \lceil c_i^{E,k} \rceil + \frac{1}{\epsilon}(\lceil c_i^{E,k} \rceil - b_i^{E,k})$. The algorithm EDF$_{1+\epsilon}$ always runs the job with the earliest deadline. The algorithm BEQUI-EDF$_{1+\epsilon}$ always broadcasts the file that EDF$_{1+\epsilon}$ is running.

The analysis of BEQUI-EDF will rely on the following lemmas. Lemma 5 proves that BEQUI-EDF is competitive as long as EDF$_{1+\epsilon}$ can complete every job by its deadline. Lemma 6 proves the well know fact that EDF$_{1+\epsilon}$ can complete every job by its deadline if any schedule with this speed can. Finally, Lemma 7 proves that some schedule with speed $1+\epsilon$ can complete the work by the deadlines because BEQUI$_1$ was able to complete the same work within the similar time constraints only shifted forward in time.

Lemma 5 For all sets of requests R, if EDF$_{1+\epsilon}$ can complete every job by its deadline, then $\frac{\text{BEQUI-EDF}_{1+\epsilon}(R)}{\text{BEQUI}_1(R)} = O(1+\frac{1}{\epsilon})$.

Proof: Consider a request $R_{i,k}$ that arrives at time $a_{i,k}$ and under BEQUI$_1$ waits until time $b_{i,k}^E$ and is serviced until time $c_{i,k}^E$. Its flow time is $[c_{i,k}^E - a_{i,k}]$. At time $c_{i,k}^E$ when BEQUI$_1$ completes this broadcast, a job $J_{i,k}^{EDF}$ is given to EDF$_{1+\epsilon}$. EDF$_{1+\epsilon}$, and hence BEQUI-EDF$_{1+\epsilon}$, complete the broadcast of the i^{th} file between the arrival time $a_{i,k}^{EDF} = \lceil c_{i,k}^E \rceil$ and deadline $d_{i,k}^{EDF} = \lceil c_i^{E,k} \rceil + \frac{1}{\epsilon}(\lceil c_i^{E,k} \rceil - b_i^{E,k})$ of this job. Hence, the completion time $c_{i,k}^{EDF}$ of this request $R_{i,k}$ under BEQUI-EDF$_{1+\epsilon}$ is before this deadline and its flow time is $[c_{i,k}^{EDF} - a_{i,k}] \leq [\lceil c_i^{E,k} \rceil + \frac{1}{\epsilon}(\lceil c_i^{E,k} \rceil - b_i^{E,k})] - a_{i,k} \leq (1+\frac{1}{\epsilon})(\lceil c_i^{E,k} \rceil - a_{i,k}) = O((1+\frac{1}{\epsilon})(c_{i,k}^E - a_{i,k}))$. (The ceilings are only needed in the case where the jobs are unit length, in which case $|c_{i,k}^E - a_{i,k}| \geq 1$.) ■

Lemma 6 will be stated in the contrapositive. If EDF$_s$ cannot complete every job by its deadline, then there is a simple local reason that no speed s schedule can, namely that there exists a time interval T such the total work that the scheduler must process during T strictly exceeds what it is capable of. More formally, given an interval $T = [u, v]$, let $\mathcal{J}_{[u,v]}$ denote the subset of jobs $J_{i,k}^{EDF}$ that must be completed completely within the interval T, because they arrive no earlier than u and have deadlines no later than time v. We say that the interval T is *over loaded* if the total work $\sum_{J_{i,k}^{EDF} \in \mathcal{J}_T} l_{i,k}$ of these jobs exceeds the amount of work $s|T|$ that the scheduler is capable of completing.

Lemma 6 If EDF$_s$ cannot complete every job by its deadline, then there exists an over loaded interval $T = [u, v]$.

Proof: Let v denote the first time v that EDF$_s$ misses a deadline of a job $J_{i',k'}^{EDF}$. Let $\mathcal{J}_{[0,v]}$ denote the subset of those jobs $J_{i,k}^{EDF}$ that have deadlines no later than time v. By the definition of EDF, whenever the job $J_{i',k'}^{EDF}$ is run in the schedule EDF$_s(\mathcal{J}_{[0,v]})$, it is also run in the original schedule. Hence, job $J_{i',k'}^{EDF}$ misses its deadline in EDF$_s(\mathcal{J}_{[0,v]})$ as well. Let u denote the last time that EDF$_s(\mathcal{J}_{[0,v]})$ idled the processor before time v. All the jobs that EDF$_s(\mathcal{J}_{[0,v]})$ processed during the time interval $T = [u, v]$ have arrival times after u (by idleness at u) and deadlines before v (by definition of $\mathcal{J}_{[0,v]}$). In other words, during interval T, EDF$_s(\mathcal{J}_{[0,v]})$ works exclusively and continuously on jobs from $\mathcal{J}_{[u,v]}$. Because EDF$_s(\mathcal{J}_{[0,v]})$ is unable to meet all of these deadlines, it follows that the total work of these jobs must exceed the amount of work $s|T|$ that the scheduler is capable of completing. ■

Lemma 7 will also be stated in the contra-positive.

Lemma 7 If there exists a time interval T during which the scheduler EDF$_{1+\epsilon}$ is over loaded, then there exists a time interval T' during which the scheduler BEQUI$_1$ is over loaded, (which is false because BEQUI$_1$ does complete its work.)

Proof: Consider some interval $T = [u, v]$ during which $\mathrm{EDF}_{1+\epsilon}$ is over loaded. We will prove that BEQUI_1 is over loaded during the interval $T' = [u - \epsilon(v-u), v]$.

Consider any job $J_{i,k}^{EDF}$ in $\mathcal{J}_{[u,v]}^{EDF}$. By definition, it is a job given to $\mathrm{EDF}_{1+\epsilon}$ whose arrival time $a_{i,k}^{EDF}$ is no earlier than time u and whose deadline $d_{i,k}^{EDF}$ is no later than time v. Now consider the k^{th} broadcast of the i^{th} file by BEQUI_1 that caused this job $J_{i,k}^{EDF}$ to be passed to $\mathrm{EDF}_{1+\epsilon}$. We will prove that this broadcast must be in the set $\mathcal{J}_{[u-\epsilon(v-u),v]}^{E}$, namely the broadcast begins at a time $b_i^{E,k}$ no earlier than time $u - \epsilon(v-u)$ and completes at a time $c_i^{E,k}$ no later than time v. The latter is simply because the broadcast completes $c_i^{E,k}$ before the corresponding job's deadline $d_{i,k}^{EDF}$, which in turn is no later than time v. The former is a little harder. Because $J_{i,k}^{EDF}$ is in $\mathcal{J}_{[u,v]}^{EDF}$, we know that the interval $[a_{i,k}^{EDF}, d_{i,k}^{EDF}]$ during which it could be run is contained within the interval $T = [u, v]$, we know that $d_{i,k}^{EDF} - a_{i,k}^{EDF} = [\lceil c_i^{E,k}\rceil + \frac{1}{\epsilon}(\lceil c_i^{E,k}\rceil - b_i^{E,k})] - \lceil c_i^{E,k}\rceil \le v - u$. Hence, $\lceil c_i^{E,k}\rceil - b_i^{E,k} = a_{i,k}^{EDF} - b_i^{E,k} \le \epsilon(v-u)$. Hence, $b_i^{E,k} \ge a_{i,k}^{EDF} - \epsilon(v-u) \ge u - \epsilon(v-u)$. This completes the proof that for each job $J_{i,k}^{EDF}$ in $\mathcal{J}_{[u,v]}^{EDF}$, the corresponding broadcast is in $\mathcal{J}_{[u-\epsilon(v-u),v]}^{E}$. The work l_i of a broadcast is equal to the work $l_{i,k}^{EDF}$ of the corresponding job $J_{i,k}^{EDF}$. It follows that the total work in $\mathcal{J}_{[u-\frac{1}{\epsilon}(v-u),v]}^{E}$ is at least the total work in $\mathcal{J}_{[u,v]}^{EDF}$.

The amount of work, $(1)(v - (u - \epsilon(v-u)))$, that BEQUI_1 is capable of completing during the interval $T' = [u - \epsilon(v-u), v]$ is equal to the amount of work, $(1+\epsilon)(v-u)$, that $\mathrm{EDF}_{1+\epsilon}$ is capable of completing during the interval $T = [u, v]$. From this we can conclude that if $\mathrm{EDF}_{1+\epsilon}$ is over loaded during the time interval $T = [u, v]$, then BEQUI_1 is over loaded during the time interval $T' = [u - \epsilon(v-u), v]$. ■

Theorem 8 The algorithm BEQUI-EDF is an $(1+\epsilon)(4+\epsilon)$-speed $O(1)$-approximation algorithm for $B|r_i, pmtn|\sum F_i$. Furthermore, BEQUI-EDF preempts each broadcast on average at most once.

Proof: The competitiveness claim follows immediately from theorem 4 and lemmas 5, 6, and 7. Note that the only reason for $\mathrm{EDF}_{1+\epsilon}$, and hence for BEQUI-EDF, to preempt is because another job $J_{i,k}^{EDF}$ with an earlier deadline arrives. It follows that the number of preemptions is at most the number of broadcasts. ■

Theorem 9 The algorithm BEQUI-EDF is an $(1+\epsilon)(4+\epsilon)$-speed $O(1)$-approximation algorithm for $B|r_i, p_i = 1|\sum F_i$.

Proof: The jobs $J_{i,k}^{EDF}$ are purposefully set to arrive at integer times. Hence, all preemptions occur at integer times. From this we know that any unit length file will never be preempted. ■

References

[1] S. Aacharya, and S. Muthukrishnan, "Scheduling on-demand broadcasts: new metrics and algorithms", ACM/IEEE International Conference on Mobile Computing and Networking, 43 – 54, 1998.

[2] A. Bar-Noy, R. Bhatia, J. Naor, and B. Schieber, "Minimizing service and operation costs of periodic scheduling", ACM/SIAM Symposium on Discrete Algorithms, 11 – 20, 1998.

[3] Y. Bartal, and S. Muthukrishnan, "Minimizing maximum response time in scheduling broadcasts", ACM/SIAM Symposium on Discrete Algorithms, 558 – 559, 2000.

[4] DirecPC website, http://www.direcpc.com.

[5] T. Erlebach and A. Hall, "NP-hardness of broadcast scheduling and inapproximatability of of single-source unsplittable min-cost flow", ACM/SIAM Symposium on Discrete Algorithms, 2002.

[6] J. Edmonds, "Scheduling in the dark", *Theoretical Computer Science*, **235**(1), 109 – 141, 2000.

[7] R. Gandhi, S. Khuller, Y. Kim and Y-C. Wan, "Approximation algorithms for broadcast scheduling", to appear at Conference on Integer Programming and Combinatorial Optimization, 2002.

[8] B. Kalyanasundaram, and K. Pruhs, "Speed is as powerful as clairvoyance", *Journal of the ACM*, **47**(4), 617 – 643, 2000.

[9] B. Kalyanasundaram, K. Pruhs, and M. Velauthapillai, "Scheduling broadcasts in wireless networks", *Journal of Scheduling*, **4**(6), 339 – 354, 2000.

[10] C. Kenyon, N. Schabanel and N. Young, "Polynomial-time approximation schemes for data broadcast", ACM Symposium on Theory of Computing, 659-666, 2000.

[11] C. Phillips, C. Stein, E. Torng, and J. Wein "Optimal time-critical scheduling via resource augmentation", *Algorimica*, **32**(2), 163 – 200, 2002.

[12] K. Pruhs and P. Uthaisombut, "A comparison of multicast pull models", manuscript.