Improving Web Performance in Broadcast-Unicast Networks

Mukesh Agrawal Amit Manjhi Nikhil Bansal
Srinivasan Seshan

July 2002
CMU-CS-02-159

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

Satellite operators have recently begun offering Internet access over their networks. Typically, users connect to the network using a modem for uplink, and a satellite dish for downlink. We investigate how the performance of these networks might be improved by two simple techniques: caching and use of the return path on the modem link. We examine the problem from a theoretical perspective and via simulation. We show that the general problem is NP-Hard, as are several special cases, and we give approximation algorithms for them. We then use insights from these cases to design practical heuristic schedulers which leverage caching and the modem downlinks. Via simulation, we show that caching alone can simultaneously reduce bandwidth requirements by 33% and improve response times by 62%. We further show that the proposed schedulers, combined with caching, yield a system that performs far better under high loads than existing systems.

This work was partially supported by the National Science Foundation, under award ANI-0092678, and by IBM Corporation.

Keywords: Satellite networks, Broadcast scheduling complexity, Web caching, Web performance, Broadcast dissemination

1 Introduction

A recent development in Internet access technologies is satellite based Internet service. In systems such as DirecPC [1] and Starband [2], users connect to the network via a satellite dish and a modem. Fig. 1 depicts a typical system. When a user visits a web page, his modem transmits the request over the telephone network to the satellite service's Network Operations Center (NOC). The NOC retrieves the web page from its cache or from the origin server over the Internet, and sends the data to a satellite. The satellite echoes the data, which is picked up by the user's satellite dish.

The appeal of these services is that satellite transmission can be much faster than modem transmission, accelerating both simple web page access and larger file transfers. However, as the satellite is a shared resource, the performance of satellite transmission decreases as the number of users increases. In contrast, the currently unused modem downlinks, while much slower than satellite, provide dedicated, or independent, channels to each user. Unlike the satellite channel, the aggregate capacity of these links scales directly with the number of users. We estimate that, in a 100,000 user system, the ratio of aggregate modem bandwidth to satellite bandwidth will be greater than 12:1[1] [3].

Given a small enough user population, and hence, a small enough service demand, it is possible to service all requests via the satellite link. As the user base grows, and during peak usage periods, the satellite is likely to become overloaded. It is important to understand when this will occur (to estimate the capacity of the system), and how to maintain performance under overload conditions.

We argue that when the system is highly loaded, performance is optimized by exploiting the fundamental properties of each type of link. The satellite link, being a broadcast channel, is well suited to the delivery of popular objects — those objects which many users are likely to want. Being a high speed channel, it is also well suited for the delivery of large objects. The modem links have the properties of being dedicated, and self-scaling. They are ideal for the delivery of small and unpopular objects. By shifting a few small objects, which still receive reasonable performance, to the modem links, we can reduce contention for the shared channel, thereby increasing capacity of the system.

As we explain below, in order to better exploit the power of the broadcast link, we require storage at the client nodes so that they can later benefit from cache hits. We note that satellite network operators already offer receivers, such as *DirecTV Receiver with TiVo* [4], with large storage capacity. Given the low and decreasing cost of disks, it is inexpensive to use some storage to cache web objects.

Fig. 1 depicts our model. The figure shows a system with a single user node; the generalization to multiple users is straightforward. In our model, each user is connected to the network via a modem, which may be used for bidirectional communication, and a satellite dish, which is capable of reception only [2]. In addition, the user node has a disk which may be used to store recently transmitted objects from both the modem and satellite channels.

Any object transmitted on the satellite link, whether requested by the node owning the cache or not, may be entered into the cache. Consider users A and B, and web page W. Suppose A loads page W at time t, and that the page is transferred via satellite. User B then places W in his cache. Later, at time $t + \epsilon$, when B accesses W, it is loaded directly from his cache. We refer to this caching of objects requested by other users as opportunistic caching, or *OpCaching*. The zipf-like distribution of web-proxy workloads [5] implies that there is sufficient locality amongst requests of *different* users for this approach to succeed.

Given this model, we seek to optimize mean response time. Response time is the time from when a user sends a request until he receives the last byte of the reply. We assume a response time of zero if the request is a cache hit in the client's cache. To accomplish this optimization task, we must answer two questions for each object transmitted. Namely, we must determine both *when* to send the data item, and *how* to send it (i.e. which link should be used). The second question is particularly significant in our setting because the links are fundamentally different.

In Sec. 2, we approach the problem from an analytical perspective. The aim is to understand how we might compute an optimal schedule offline, given complete knowledge of the series of user requests. Based on the theoretical analysis, Sec. 3 proposes and evaluates offline heuristics. This offline evaluation isolates

[1] We estimate that DirecPC dedicates ten 45 Mb/sec satellite channels to its roughly 100,000 users.

[2] We use modem and unicast interchangeably. We also use satellite and broadcast interchangeably. We also use document, file, object and response interchangeably.

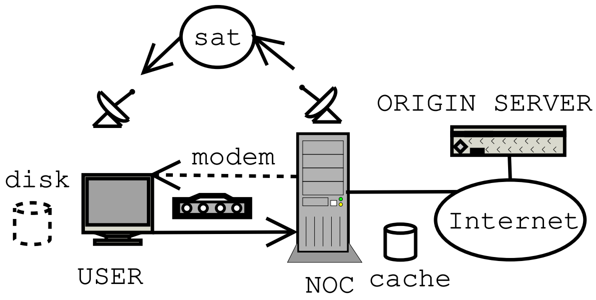

Figure 1: Satellite based Internet access system. Links are bidirectional unless otherwise indicated. Dashed lines indicate elements not exploited in existing systems.

heuristic performance from predictor accuracy. Next, in Sec. 4, we use promising aspects of the offline heuristics to develop online schedulers. Our performance evaluation of these online schedulers shows that clever scheduling, coupled with OpCaching, improves both scalability and response time under high loads.

Specific contributions of this work are:

- Analyzing the problem of scheduling the combination of broadcast and unicast channels. We also analyze simplified variants of this scheduling problem. We give approximation algorithms for the variants that we show to be NP-Hard. For several other special cases, we give polynomial time algorithms.
- Proving a lower bound of $\Omega(\sqrt{n})$ on the competitive ratio for any online algorithm.
- Introducing opportunistic caching, which can reduce the bandwidth required to service peak loads by 33%, while improving mean response time by 62%.
- Describing schedulers that gracefully degrade performance during high load periods. The schedulers simultaneously achieve lower response times, and complete more requests than the strategy of using the satellite alone.
- Demonstrating that, with typical web workloads, it is possible to accelerate large object access, while providing good performance for small objects, even at peak loads.

2 Theoretical section

In this section, we analyze our problem from a theoretical perspective[3], assuming an infinite cache at the client's end. Therefore, no file needs to be broadcast more than once. We gradually build from a simple scenario, where only the broadcast downlink is used and all documents are of the same size, to the most realistic scenario, which includes broadcast and unicast links and variable sized objects. We investigate the problem of finding *optimal offline* [4] solutions for each scenario. Table 1 summarizes the results. In the last subsection, we consider online algorithms and prove that no $o(\sqrt{n})$-competitive algorithm exists for either the broadcast-unicast case, or its variants.

We show most of the scenarios to be NP-Hard, and give approximation algorithms for them. Our algorithms use *resource augmentation*, a paradigm introduced by Kalyanasundaram and Pruhs [6] and widely used in scheduling literature. The idea is to allow our algorithm a slightly faster $(1+\epsilon)$ speed processor. We then compare our solution to the optimum solution, but the optimum is only allowed to use a unit speed

[3] Formally, we explore minimizing:

$$\sum_{j \in Clients} \sum_{i \in Files} \max((res_{ij} - req_{ij}), 0)$$

where req_{ij} is the time when client j requests object i (∞ if no such request is made), and res_{ij} denotes the time when the client receives the document (either by the broadcast link or the unicast link, whichever is first).

[4] Offline means that the NOC *a priori* knows the request dates for all objects, i.e. it has complete future knowledge.

Table 1: Summary of theoretical results

Downlink	Files	Results	Section
Broadcast	Same-size	$O(n^3)$	2.1
Broadcast	Variable-size	NP-Hard, $(1+\epsilon)$-speed, $(1+\frac{1}{\epsilon})$-approx	2.2
Broadcast	Uniform popularity	3-approx, optimal (special case)	2.3
Broadcast + Unicast	Same-size	NP-Hard	2.4
Broadcast + Unicast	Variable-size	$(1+\epsilon)$-speed,$(1+\frac{1}{\epsilon})$-approx	2.4

processor. We say that an algorithm is $(s - speed, c - approximate)$ if it uses an s times faster processor than the optimal algorithm and produces a schedule which is no more than c times worse than that of the optimum. Note that in our scenario providing a speedup of s translates to increasing the bandwidth of a link by a factor of s.

To obtain a more realistic model, we can associate a *release date* r_i with each file i. This denotes the time when the document is first available to be sent. It can be used to account for the time it takes for the NOC to fetch the document from the origin server or the fact that a document (such as the score of a basketball match) is only available after a certain time. Most of the techniques we present in this section can be extended to incorporate release dates. We can also envision a scenario in which the NOC has more than one satellite downlink at its disposal, all of which are equivalent (users *OpCache* files sent on either of the links). We provide details of these extensions wherever necessary.

2.1 Same sized documents - broadcast only

We first consider the simplest case when there is a single broadcast channel, all files have unit size, and requests have arbitrary weights. We show that this problem can be solved by computing a minimum cost perfect matching in a suitably constructed graph.

Let m denote the number of unique documents (files). For each file i, let c_{ij} denote the cost incurred if the file is served at time j. That is, c_{ij} equals the sum of the weighted response times incurred by clients if file i is sent at time j. We construct a bipartite graph with m nodes on the left, where each node corresponds to a document, and m nodes on the right, where each node corresponds to a unit time interval. The edge ij between node i on the left and node j on the right has cost c_{ij}. Finding a schedule now corresponds to finding a minimum cost matching in the graph, which can be solved in $O(m^3)$ [7]. If there are release dates r_i for each document i, the same method applies by setting c_{ij} to ∞ for $j < r_i$.

For the case of multiple broadcast channels, $b > 1$, we modify the construction as follows. Let $T = \lceil \frac{m}{b} \rceil$. Clearly all files will be broadcast by time T in the optimal schedule. Let us label the channels $0, \ldots, b-1$. We now create a complete bipartite with m nodes on the left and bT nodes on the right, where c_{ij} denotes the cost of sending file i on channel $\lfloor \frac{j}{T} \rfloor$ and time $j - T\lfloor \frac{j}{T} \rfloor$. Every matching in this graph corresponds to a valid schedule and vice-versa. Thus, the result follows. Again, if we have release dates, the bipartite graph can be modified suitably.

2.2 Different sized documents – broadcast only

Unlike the case of unit size documents, scheduling different sized documents on a broadcast link is NP-Hard. We show that this case is a generalization of the *minimum weighted tardiness*[5] problem, which is NP-Hard [9]. We also give an approximation algorithm for this problem.

For multiple channels, the hardness follows trivially from a reduction from the Partition problem [8]. Given a set of numbers $s_1, \ldots, s_n$ with total sum S, the partition problem asks whether the numbers can be divided into two parts where the sum of each part is exactly $S/2$. Now consider the case where we have 2 broadcast channels, we can simply have documents with sizes $s_1, ..., s_n$, such that each document is requested

[5]Given a set S of tasks all released at time 0, let $s(i)$, $w(i)$ and $d(i)$ denote the processing time (size), weight and deadline respectively of job i. Given a schedule, the tardiness of a job is defined as the amount by which it exceeds its deadline (the tardiness is 0 if it finishes before its deadline). Find a single processor schedule that minimizes total weighted tardiness [8].

at time $S/2$. Observe that the optimal broadcast schedule has cost zero if and only if there exists a partition for the original problem.

Proving NP-Hardness: For each job i with size, weight and deadline s_i, w_i and d_i in the weighted tardiness problem, let there be a corresponding document in our scheduling problem (call it i), which has size s_i and for which w_i requests arrive at time d_i. In any schedule, the weighted tardiness of file i is exactly identical to the weighted response time in our broadcast schedule. Thus, our problem is at least as hard. It is interesting to note that the weighted tardiness problem remains NP-Hard even if all the weights are one [10]. This corresponds to the fact that our broadcast scheduling is hard even if each file has just one request.

Notice that if we have release date constraints, and if w_i users send requests for file i at time r_i, then computing an optimal broadcast schedule is identical to the problem of minimizing the weighted response time on a single machine. Thus, our problem generalizes the *weighted response time* problem, for which no $o(n)$ approximation algorithms are known (except for special cases). Thus, it seems hard to find a good approximation algorithm that does not use speedup for our more general single channel broadcast problem. Instead, we give a speedup-based approximation algorithm.

A ((1+ϵ)-speed, (1+$\frac{1}{\epsilon}$)-approximate) Algorithm: First observe that the response time in the broadcast schedule can be thought of as $\sum_{i \in Files} \sum_{t \leq c_i} o_{i,t}$, where $o_{i,t}$ the number of outstanding requests for file i at time t and c_i is the time when file i is completely transmitted. Consider the *fractional response time* metric, where instead of counting the total number of outstanding requests for file i at time t, we consider the fraction of the file remaining at time t times the number of outstanding requests. Thus, if a third of file i is sent by time t we count $2/3 o_{i,t}$ for that time in our metric. Clearly, for any schedule the value of the fractional response time metric is no more than the value of the original response time metric.

Let *Opt* be the schedule which minimizes the value of the total response time, $V(Opt)$ denote its value, and $F(Opt)$ denote its fractional response time. Thus, we know that $F(Opt) \leq V(Opt)$. Now let Opt' be the schedule which minimizes the fraction response time. Thus, $F(Opt') \leq F(Opt) \leq V(Opt)$.

Let us assume for now that the file can be broken up into infinitesimally small chunks. Then Opt' can be computed optimally, since it simply reduces to the problem of minimizing the broadcast schedule with unit size objects with arbitrary weights. More precisely, to transform the original problem, if there was a request for a file of size s_i, we break it up into s_i requests, one for each chunk of the file. Each chunk of the file has weight $1/s_i$ and size 1.

Having obtained Opt', we observe that if we have a $(1+\epsilon)$ speed processor and just mimic the schedule of Opt', then at the time when we finish sending file i, Opt' would still have a $\beta = \frac{\epsilon}{1+\epsilon}$ fraction of the file left. Thus, if we have a faster processor then the cost of schedule Opt' using the original metric, $V(Opt')$, can be no more than $\frac{1}{\beta} = 1 + \frac{1}{\epsilon}$.

Thus, we have a ($1+\epsilon$-speed, $1+\frac{1}{\epsilon}$-approximate) algorithm. Recall, we assumed that a file can be broken into very small chunks. However, simply breaking it into chunks of size $c\epsilon$, where $c < 1$, suffices. This affects the approximation negligibly.

Making the Schedule Non-preemptive: We observe that the schedule returned by Opt' might schedule various chunks of the same file in non-contiguous time blocks. However, it can be made non-preemptive. Non-preemptive schedules are preferable in real systems, because there might be overheads associated with preemption. To make the schedule non-preemptive, we simply consider the times when $\frac{\epsilon}{1+\epsilon}$ fraction of a file in Opt' is remaining. Call this time t_i for file i. We now order the files according to increasing t_i and broadcast them in this order using our $1+\epsilon$ faster processor. The above procedure clearly produces a feasible non-preemptive schedule.

2.3 Different sized documents, uniform arrivals over time — broadcast only

Although a non-speedup approximation algorithm for the general case of scheduling variable sized objects on a broadcast link is unlikely, such algorithms are possible for important special cases of the problem. For the case where the request rate, λ_i, is uniform over time for all objects i, we give a (1-speed, 3-approximate) algorithm. For the further constrained case where the request rate of an object is correlated with its size, s_i, such that the *density* of objects, λ_i/s_i, satisfies $\frac{\lambda_i}{s_i} > \frac{\lambda_j}{s_j} \iff \lambda_i > \lambda_j$, we show that scheduling in order of decreasing density is optimal.

Note that if the file reaches the clients at time t over the broadcast channel, then the total expected waiting time for the outstanding requests for the file is $\int_0^t (t-x)\lambda_i dx = \lambda_i t^2/2$. Thus, the problem of minimizing the total response time in this case is equivalent to minimizing the weighted response time squared where all the release dates are 0, and the weight of the file is given by λ_i.

A (1-speed, 3-approximate) Algorithm: Transmitting the files in non-increasing order of λ_i/s_i gives us the required algorithm. Let r_i denote the response time of file i and let *Opt* denote the optimal value of $\sum_i \lambda_i r_i^2$. We consider another metric which we call the *fractional weighted response time squared*. In this metric we divide file i into s_i chunks each of size 1 and weight λ_i/s_i each. If r_{ij} denotes the response time of the j^{th} chunk of file i, the value of the metric is $\sum_i \sum_{j=1}^{s_i} \frac{\lambda_i}{s_i} r_{ij}^2$. Clearly, for any schedule, the value of fractional weighted response time squared is at most that of weighted completion squared metric. Let F_{Opt} denote the fractional weighted response time squared for the schedule *Opt*.

Let $Opt(F)$ denote the cost of a schedule which minimizes the fractional weighted response time squared. Clearly, $Opt(F) \leq F_{Opt} \leq Opt$. Now, finding the optimal fractional schedule is easy, since each job is of size 1 and weight λ_i/s_i. Simply scheduling the jobs according to non-increasing order of λ_i/s_i minimizes the fractional weighted response time squared. We show that the optimal fractional response time schedule yields a solution that is no more than 3 times worse than the optimal traditional response time schedule. This gives our desired 3-approximation.

Let r_{i0} denote the starting position of the first chunk in the optimal fraction schedule. Thus, the contribution of file i to the fractional weighted response time squared is $\lambda_i/s_i \sum_{j=1}^{s_i}(r_{i0}+j)^2$ whereas the contribution to the original metric is $\lambda_i(r_{i0}+s_i)^2$. Note that

$$\begin{aligned}
& \lambda_i/s_i \sum_{j=1}^{s_i}(r_{i0}+j)^2 \\
= \;& \lambda_i(r_{i0}^2 + r_{i0}(s_i+1) + (s_i+1)(2s_i+1)/6) \\
\geq \;& \lambda_i(r_{i0}^2 + r_{i0}s_i + s_i^2/3) \\
\geq \;& \frac{1}{3}\lambda_i(r_{i0}+s_i)^2
\end{aligned}$$

Thus, the 3-approximation follows.

We observe that the schedule produced by the algorithm above has the advantage of being non-preemptive. If the file sizes and weights are small compared to the time horizon, it can be show that the schedule produced is close to optimum.

The above method works for any n ($n > 0$). If instead of $\lambda_i t^2$, the response time metric is $\lambda_i t^n$, the above method gives a $(n+1) - \textit{approximate}$ algorithm ($\forall\ n \geq 2$).

An Optimal Algorithm: When the order of $\frac{\lambda_i}{s_i}$ is the same as λ_i, scheduling in decreasing order of λ_i/s_i is optimal. We show that this holds whenever the response time metric is λt^n, for any positive n. Assume to the contrary. Then, there exist 2 consecutive jobs, say job 1 and job 2, in an optimal schedule *OPT*, such that 2 is scheduled before 1, even though $(\frac{\lambda_2}{s_2} < \frac{\lambda_1}{s_1})$. We will show that switching them improves the metric, and so, we get a contradiction to optimality.

Note that switching 1 and 2 does not affect the schedule of any jobs except these two (since these are consecutive jobs). Let the schedule obtained by switching 1 and 2 be *OPT'* and the contribution of response time from job 1 and 2 be R_{OPT}.

Let x be the time when job 2 starts in *OPT*. Then,

$$\begin{aligned}
R_{OPT} &= \lambda_2(x+s_2)^n + \lambda_1(x+s_1+s_2)^n \\
R_{OPT'} &= \lambda_1(x+s_1)^n + \lambda_2(x+s_1+s_2)^n
\end{aligned}$$

We show that

$$\lambda_2(x+s_1+s_2)^n - \lambda_2(x+s_2)^n \leq \lambda_1(x+s_1+s_2)^n - \lambda_1(x+s_1)^n$$

and hence, interchanging the jobs leads to a better schedule.

If $s_1 \leq s_2$, we are directly done, since $\lambda_2 < \lambda_1$.

If $s_1 > s_2$, we rewrite as

$$\lambda_2 s_1 (\sum_{i=0}^{n-1} (x+s_1+s_2)^i (x+s_2)^{n-1-i}) \leq \lambda_1 s_2 (\sum_{i=0}^{n-1} (x+s_1+s_2)^i (x+s_1)^{n-1-i})$$

Now, since $\lambda_1 s_2 > \lambda_2 s_1$, and the right hand side summation is term-by-term larger than the left hand side summation, it follows that OPT' is better than OPT. Thus, the contradiction is shown.

2.4 The Broadcast-Unicast problem

We now explore the target problem of scheduling for both the broadcast and the unicast channels. Unfortunately, we show that the Broadcast-Unicast problem is NP-Hard, even if all files are unit-sized. Although we give an approximation algorithm, it provides few insights into devising a practical algorithm.

Proving NP-Hardness: We give a transformation from *Exact Cover by 3-Sets*[6], in which each element occurs in at most 3 subsets [8].

Given an instance of *Exact Cover by 3 sets*, let the elements of X be $u_1, \ldots, u_{3q}$ and let $S_i \in C$ denote the 3-element subsets. We create an instance of the Broadcast-Unicast problem as follows: Each u_i corresponds to a client and each S_i corresponds to a file. Client u_i requests file j iff $i \in S_j$. Note that since each element lies in at most 3 sets, each client requests at most 3 files. Next, we add dummy requests for each client such that total number of files requested by a client is exactly $q+1$. A dummy request for a client u_i is a file which is requested by u_i only. Finally, all the $q+1$ requests by each client (u_1 through u_{3q}), are made at time $t = q$. Our goal is to show that all the requests can be satisfied by time q (i.e. the Broadcast-Unicast problem has cost zero) iff the original problem instance has an exact cover.

First, we show that given an exact cover C' we can satisfy all requests by time q. This is done by transmitting file i for each $S_i \in C'$ over the broadcast channel. By the definition of exact cover, the broadcast schedule will satisfy exactly one request for each of the $3q$ clients. Finally, for each client we simply send the remaining q requests using its personal unicast link. Thus, all requests are satisfied by time q.

For the other direction, we show that any zero cost broadcast-unicast schedule gives an exact cover. Since all requests are satisfied by time q, each client has at least one request satisfied by the broadcast channel. Now, any file transmitted by the broadcast channel can either satisfy 3 clients (if it is a file corresponding to some S_i) or 1 client (if it is a dummy file for some client). Since there are $3q$ clients and each has at least one request satisfied by the broadcast channel, every file sent over the broadcast channel must correspond to some S_i. Thus, we have an exact cover in the original instance.

Hence, finding the optimal Broadcast-Unicast schedule is NP-Hard. Moreover, it is also impossible to obtain a non-speedup approximation algorithm for the problem (since any algorithm which is not optimal has a non-zero cost, whereas the optimal cost is zero). Although no 1-speed approximation is possible, a $(1+\epsilon)$-speed, $(1+\frac{1}{\epsilon})$-approximate algorithm is possible using linear programming. We give the details below.

A (($1+\epsilon$)-speed, ($1+\frac{1}{\epsilon}$)-approximate) Algorithm: The idea is to obtain a schedule which is optimal for the fractional response time metric. However, in the case of both unicast and broadcast, there is no easy combinatorial algorithm for optimizing the fractional response time. We obtain the fractional optimum using a time-index linear programming formulation. We then use the schedule obtained and show that with a $(1+\epsilon)$ speed-up in the capacity of the both the unicast and broadcast channels we can obtain a schedule whose cost is no worse than $(1+1/\epsilon)$ times the optimum with respect to the (non-fractional) weighted response time.

Let s_i, r_i denote the size and release time of file i. Let $w_{j,i}$ be the weight that user j assigns to file i. $b_{i,t}$ will be denote the amount (bytes) of file i transmitted over the broadcast link at time t. Let c_j denote the bandwidth of the unicast channel of user j. Let C denote the bandwidth of the common broadcast link. Let $u_{j,i,t}$ denote the amount of file i sent to user j at time t over the unicast channel.

Consider the following linear program:

[6] Given a set X with $|X| = 3q$ and a collection C of 3-element subsets of X. Does there exist an exact cover for X, i.e., a sub-collection $C' \subseteq C$ such that every element of X occurs in exactly one member of C'? The problem is NP-Complete even if no element occurs in more than three subsets.

$$Minimize \sum_{j \in Clients} \sum_{i \in Files} \sum_{t=0}^{T} w_{j,i} rem_{j,i,t}$$

Subject to

$$\begin{aligned}
\sum_{i \in Files} b_{i,t} &\leq C \qquad \forall t & (1)\\
\sum_{i \in Files} u_{j,i,t} &\leq c_j \qquad \forall j,t & (2)\\
\sum_{t=0}^{T} b_{i,t} + u_{j,i,t} &\geq p_i \qquad \forall j,i & (3)\\
rem_{j,i,t} &= 1 - \frac{1}{p_i}\left(\sum_{x=0}^{t-1}(b_{i,x} + u_{j,i,x})\right), \forall j,i,t & (4)\\
rem_{j,i,t} &\geq 0 \qquad \forall j,i,t & (5)\\
b_{i,t} &\geq 0 \qquad \forall i,t & (6)\\
u_{j,i,t} &\geq 0 \qquad \forall j,i,t & (7)
\end{aligned}$$

Constraints 1 and 2 ensure the bandwidth constraints for the broadcast and the unicast channels. Constraint 3 ensures that all the bytes of a file are transmitted by the schedule. Constraint 4 defines the fraction of the file remaining by time t. The last three constrains are positivity constraints. The objective function measures the fractional weighted response time of the broadcast-unicast schedule.

Having obtained the optimal LP-solution (the LP can be solved in polynomial time using the ellipsoid algorithm [7]), we obtain a schedule as follows: starting from time $t = 0$, whenever the LP solution sends $b_{i,t}$ units of file i by t, we send $b_{i,t}(1+\epsilon)$ units of i our faster processor. On the broadcast channel, we start sending the file from its beginning, i.e., we send the first $(1+\epsilon)b_{i,0}$ units at time 0, next $(1+\epsilon)b_{i,1}$ units at time 1 as so on. For deciding what to send on the unicast channel we do the following: for a file i and user j, let $t(i,j)$ denote the earliest time when $1/(1+\epsilon)$ fraction of the file has been sent by the LP, i.e. $rem_{j,i,t(i,j)} \leq \epsilon/(1+\epsilon)$. Let $b_{total}(i,t) = \sum_{x=0}^{t-1} b_{i,x}$. On the unicast channel, beginning from time $t = 0$, we start sending the file i starting from the unit $(1+\epsilon)b_{total}(i, t(i,j))$. Notice that using this procedure, every user j will have the file i by time $t(i,j)$. Finally, arguing as in Sec. 2.2, it can be seen that the solution obtained is no more than $(1 + 1/\epsilon)$ times the optimum.

Note that in the above schedule, a part of the file may be sent via broadcast, and the remaining part might be sent via unicast. If we want to ensure that a file reaches completely via a single channel, we can do this if we have a $(2+\epsilon)$ speedup, $\epsilon > 0$. Note that for every user j, there exists time $t(i,j)$ such that at least $1/(2+\epsilon)$ fraction of file i either reaches by unicast or by broadcast. Now, if both our unicast and broadcast are sped up by a factor of $(2+\epsilon)$, the complete file will reach by time $t(j,i)$. Again rounding the LP solution as above, it can be seen that this gives us a $(1+\frac{2}{\epsilon})$-approximate schedule.

This solution is applicable even if the unicast channels have different bandwidths, and we need to consider release dates.

2.5 Online Algorithms

In this subsection, we consider how well online algorithms perform as compared to their offline counterparts in our scenarios. An algorithm is said to be c-competitive if it is a c-approximation algorithm that computes online.

Because an *offline* algorithm can use the cache in a more judicious fashion by pushing documents that it knows would be requested next, it can perform considerably better than an *online* algorithm. We prove that no $o(\sqrt{n})$-competitive algorithm exists, even for the simple case in which just one broadcast channel is available for sending files, and the files are same-sized. This result holds even if we make changes to this model like adding back the unicast channels, having different sized-objects, and having multiple broadcast channels.

Consider $2k$ unit length files that are each requested at time $t = 0$. At time k, there are k additional requests for each of the k remaining files. The document sent in slot $[k + i - 1,\ k + i)$ would contribute $i * (k + 1) + k$ to the total response time. Thus, the total response time for the online schedule is $\Theta(k^3)$. The offline scheduler would send out the k popular ones first, and hence, the total response time would be $\Theta(k^2)$. The total number of requests n=$\Theta(k^2)$, and hence, the competitive ratio is $\Theta(\sqrt{n})$. The above also shows that if no more than k requests arrive at any time, then a lower bound of $\Omega(\sqrt{k})$ holds.

3 Offline Heuristics

Since finding an optimal solution to the offline broadcast-unicast problem is NP-Hard (Sec. 2.4), in this section, we devise and evaluate heuristics to minimize the average response time. Our primary focus is to find heuristics that work well in an online setting, since our algorithms would finally be deployed in an online setting. In addition, we also evaluate heuristics that would work well in an offline setting, to get a sense of how much better we could perform if we had a perfect predictor. Subsection 3.1 details our heuristics. In Subsection 3.2, we present the results and expound on the lessons learned. These in turn motivate our choices for the online algorithms evaluated in Sec. 4.

3.1 Heuristics

Instead of solving the whole problem in one step, we solve the broadcast case first, and then use it to determine the unicast schedule. This is a natural approach, since the broadcast channel has much higher bandwidth and also happens to be shared, thus making it imperative to find a good schedule for it. Fortunately, we also have some heuristics that are guaranteed to work well in the *broadcast-only* case (Sec. 2.2, 2.3).

Note that the broadcast schedule tells us the time when a particular file would reach the clients. For determining each client's unicast schedule, we then step through its requests in order of their arrival time, and send a requested file over the unicast link iff doing so makes the new waiting time of the client for that particular request *threshold* times the original waiting time[7]. Reducing *threshold* involves a tradeoff — forsaking immediate gains (reduction in waiting time due to the present request) in anticipation of future gains (subsequent documents sent over unicast would reach the user faster)[8]. We experiment with different values of *threshold* in arriving at a unicast schedule.

Now that we have reduced our problem to finding a good broadcast schedule, we can use insights from Sec. 2 to devise a broadcast schedule that works well. From Sec. 2.2, we know a $(1 + \epsilon) - \mathit{speed}, (1 + 1/\epsilon)$-approximate algorithm that uses the technique of dividing each file into chunks of equal size, and basing the broadcast schedule on the optimal order of sending the equal sized chunks (Sec. 2.1). Unfortunately, the $O(m^3)$ matching algorithm (Sec. 2.1) does not scale. Moreover, in our preliminary evaluation on smaller data sets, our geometric scheduler (described below) performed equally well. So, we do not use this heuristic any further in our experiments. One other heuristic that works well, when files are uniformly popular, is to schedule in order of densities (Sec. 2.3). We generalize the density function (popularity/size) to $\frac{\lambda_i}{s_i^\alpha}$ and experiment with different values of α between 0 and 1. We do this because $\alpha = 1$ does not seem to be the optimal choice when the two alternative links have widely different bandwidths[9] and capabilities (broadcast

[7] If a request is satisfied by unicast, then the number of outstanding requests for the file decreases, and this may invalidate the broadcast schedule already computed. Thus, a more elaborate but accurate mechanism that we did not use, would be to re-compute the broadcast schedule whenever a request is satisfied by unicast. This could potentially make the new running time of any heuristic, the old running time times the total number of requests.

[8] Note that the problem of finding the optimal solution here is NP-Hard. The problem can be formulated for each client as given b_i, req_i, find a unicast schedule such that $\sum_{i \in Files} \max((\min(b_i, \sigma_i) - req_i), 0)$ is minimized, where b_i and σ_i are the time instants when the file reaches the user via broadcast and unicast channels, and req_i is the time when the user requested file i. If all files are same-sized, the problem can be solved by min-cost bipartite matching, else it is NP-Hard.

Showing NP-Hardness: Assume a user, for whom the b_i values are very large. Then, this problem is the same as minimizing weighted tardiness with all weights as 1 (Sec. 2.2), where the deadline for file i is req_i, and its release date is 0.

[9] Assume a simplistic case, in which 2 files, one of size 900KB (requested by 900 users), and the other of size 1KB (requested by 2 users) need to be scheduled. All requests arrive at time t=0. If $\alpha = 1$, we send the smaller file first on broadcast channel. Note that the unicast channels in this case remain idle, since the larger file, even though sent second on the broadcast link, reaches the user faster than the unicast link (because the unicast link is extremely slow). If $\alpha = 0$, the smaller file would be sent over the unicast links and the total response time could be reduced.

Table 2: Evaluating heuristics: Avg. response time (normalized)

Size	Naive (thres=0.55)	Geometric ($\alpha = 0$)	Sorted ($\alpha = 0$)	Sorted ($\alpha = 1$)	Naive (thres=1)
5MB	1.00	0.89	0.95	1.20	1.04
10MB	1.00	0.94	0.98	1.18	1.02
20MB	1.00	0.94	0.97	1.15	1.02
50MB	1.00	0.92	0.94	1.14	1.02

Table 3: Avg. response time (normalized) vs. threshold, Size=50MB

Threshold values	0.10	0.30	0.55	0.80	1.0
Naive	1.20	1.00	1.00	1.02	1.02
Sorted($\alpha = 0$)	1.28	0.98	0.94	0.93	0.94
Geometric($\alpha = 0$)	1.23	0.94	0.91	0.92	0.93

vs. unicast). Also, in the presence of OpCaching, sending popular objects over the broadcast channel is likely to improve performance by enabling cache hits, thus strengthening the case for a lower α value.

We employ the following heuristics to find the *broadcast-only* schedule:

Naive Scheduler This simple heuristic serves files in first-come, first-serve order. A file's position in the broadcast schedule is dependent on the time when it is first requested by any user.

Sorted Scheduler This scheduler is based on $\frac{\lambda_i}{s_i^\alpha}$. For this offline scheduler, we consider λ_i to be the total number of requests that arrived for file i during the whole time period.

Geometric Scheduler This is a refined version of the above, paying attention to both the request arrival times, and popularity. Here, we divide the time period into small windows, consider requests for a file arriving in different windows separately, and take a weighted sum of these numbers[10]. Also, note that this offline scheduler has a computational cost of $0(m^2)$. We tune the geometric scheduler's performance by experimenting with different window sizes, to get an idea of the best we could perform, if we had enough computational resources and access to a perfect predictor (larger window sizes work better with larger traces). Finally, there seems to be no easy adaptation of this heuristic to the online scenario because it requires fine grained predictions. Moreover, this and Sorted would map to the same online algorithm, if we used a limited look-ahead.

3.2 Results

Using the two step process (Sec. 3.1), we find the average response time across all requests, assuming an infinite cache at each client. Our simulation includes client request queues, response queues, transmission delays and link latencies (Table 5).

We evaluate our heuristics on traces from the Polygraph workload generator (Sec. 4.2). Our traces consist of requests for variable sized files. We define the *size* of a trace to be the total size of all unique objects requested in the trace. We use traces of size 5MB, 10MB, 20MB, and 50MB for our evaluation. We run our algorithms on two different traces for each size, and present the average of the two results in Table 2. We use $threshold = 0.55$ unless specified.

We make the following observations:

[10] Specifically, we calculate a *score* for each object i at the beginning of each time window j,

$$score_{ij} = \sum_{k=0}^{k=j-1} req_{ij} + \sum_{k=j}^{k=n} \frac{req_{ij}}{ratio^{k-j-1}}$$

where req_{ij} is the number of requests received for object i divided by s_i^α in the time window j, and $ratio$ is a number greater than 1 to weigh down the future requests. Empirically, we found $ratio = 1.1$ to work well, and use it for our evaluation.

1. The Geometric scheduler is better than Sorted ($\alpha = 0$), which in turn is better than Naive (Table 2). This is in line with our expectations, based on the complexity and requirement of each heuristic. We believe that the reason Naive performs only about 8% worse is because the popularity of files does not change very rapidly either in real-life workloads, or in our traces.

2. The response time of any heuristic varies in a paraboloid fashion, as the *threshold* is increased from 0 (unicast links not being used) to 1 (greedy choice), with a minima close to $threshold = 0.55$ (Table 3). This means that a heuristic which tries to balance the load on the unicast and broadcast links is likely to perform well. A more popular object, instead of being sent by unicast (unless it is substantially faster), should be sent via broadcast, so that it can later result in cache hits, and so as to reduce load on the unicast links. Our GPop scheduler (Sec. 4.1) considers both the popularity and the greediness for determining how and when to send an object.

3. For Sorted, the response time decreases with decreasing α, and we obtain the best result for $\alpha = 0$. As a result, we do not consider the *size* of objects in any of the *online* algorithms in the next section.

4 Simulation

In this section we evaluate the ability of several online heuristic schedulers to improve mean response time for web access. We first describe the proposed schedulers. Next, we detail our evaluation methodology. We then present our experimental results.

4.1 Practical Schedulers

We consider three scheduling heuristics: *Greedy*, *Sorted*, and *GPop*. Greedy and Sorted represent online adaptations of the Naive and Sorted offline schedulers respectively. GPop is a hybrid. We omit the Geometric scheduler of Sec. 3 as it has no online equivalent.

Like the Naive scheduler, Greedy ignores popularity and defaults to servicing requests via broadcast. Because an online scheduler cannot employ the two-step process of Sec. 3, Greedy employs an alternate method of shifting replies to the unicast links. On receiving a request, Greedy estimates the service time over both links by assuming fair sharing of links by all requests enqueued on the link, and computing $size \times queuelength/bandwidth$. If the satellite is sufficiently congested that the modem link is faster, Greedy sends the reply via modem. We also experimented with a variation of Greedy, *GreedyDelay*, that includes link latency in the estimated service time. We observed that GreedyDelay performed worse than Greedy because GreedyDelay is more likely to send small objects, which tend to be more popular, via modem than Greedy. We thus omit detailed results for GreedyDelay.

The online form of Sorted sends every reply via unicast. Sorted maintains popularity information using a counter r_o for each object o. Initially, $r_o = 0\ \forall o$. Each time o is served via unicast, r_o is incremented. Any time o is served via broadcast, r_o is reset to zero. Periodically, Sorted pushes the most popular objects to users via the broadcast link, in order of popularity. Through simulations, we have found that a period of one second gives the best results.

GPop embodies the lessons learned in Sec. 3 by incorporating popularity, and attempting to avoid overusing any link. In deciding whether to service a request via broadcast or unicast, GPop considers the service time on both links, as in Greedy. To incorporate popularity, GPop maintains popularity in the same manner as Sorted. GPop serves a request over the broadcast link if $time_{bcast} \leq time_{ucast} \times satpref \times r_o$. The $satpref$ parameter enables us to tune the behavior of GPop. We have found 0.06 to be a good value for $satpref$[11].

4.2 Methodology

We simulate the performance of the system running against synthetic web workloads. In order to properly model client access patterns across web sites, we focus on proxy workloads rather than server workloads. The simulations are conducted with an extended version of the *ns-2* [11] network simulator which incorporates

[11]The Appendix provides details on tuning *satpref*.

Table 4: Typical workload characteristics

Parameter	Value
Simulation length	8 hours
Warmup period	4 hours
Mean request rate	0.4/client/sec
Mean requested file size	7KB
Max requested file size	2MB
# of users	1000

Table 5: Link characteristics

	One-way Latency	Bandwidth
Satellite	125ms	2.25 Mbits/sec
Modem	50ms	56 Kbits/sec

portions of the *Polygraph* [12] proxy benchmarking tool. The Polygraph benchmark was developed for the annual *Cache-off* s[13].

To the extent possible, we use the stock *polymix-4* workload from the Polygraph package. The aim of the workload is to gauge how well the system performs under typical peak daily loads. One significant modification is that we omit the multi-phase schedule of Polygraph for a shorter schedule. As we take our measurements after the system has warmed up, we expect that the altered phase schedule does not significantly alter the results.

We give characteristics of a typical Polygraph run in Table 4. Individual runs deviate slightly from these nominal values. Table 5 gives the nominal characteristics of the modem and satellite links. Links are scheduled with fair sharing[12]. Based on the analysis of [14], we choose the satellite bandwidth for our 1000 user system to be 2.25 Mb/sec.

Our simulations omit Internet and server delays. The client nodes enter every broadcast object into their caches, and employ an LRU eviction policy.

4.3 OpCaching Benefits

We first consider the benefits of OpCaching. To isolate the benefits of OpCaching from the benefits of using the downlink channel of the modem links, we examine the setting where only the satellite downlink is used. We measure the reduction in minimum bandwidth required to meet user demand both with and without OpCaching.

To determine the minimum bandwidth, we run simulations with varying satellite bandwidth. We start with a high bandwidth setting and progressively reduce bandwidth until the system is saturated, as indicated by a large queue on the satellite link. This procedure is run both with and without caching at the client nodes.

Table 6 summarizes the results. The "Mean Resp" column gives the response time averaged over all requests, with the time for a cache hit being zero. The "Mean Xfer" column gives the average response time for cache misses only. Boldfaced entries denote configurations where the system failed to complete all client requests.

We find that, without OpCaching, the system requires a minimum bandwidth of 21 Mb/s to meet user demand. Increasing bandwidth beyond this point does not improve response time significantly, as overheads such as the time to send the request comprise a large fraction of the response time in this setting. With OpCaching, we can reduce bandwidth by 33%, to 14 Mb/s, while improving mean response time by 62%, from 0.95s to 0.36s.

As an aside, it might seem surprising that a system with a 14 Mb/s link that employs OpCaching achieves lower response times on *cache misses* than a system with a 35 Mb/s link that does not use OpCaching. Given

[12]The exception to this is that the Sorted scheduler sends objects on the broadcast link in strict popularity order.

Table 6: OpCaching Benefits

Cache	BW (Mb/sec)	Mean Queue	Mean Resp (sec)	Mean Xfer (sec)
none	**20**	**3825**	**158.34**	**158.34**
none	21	55.3	0.95	0.95
none	22	22.6	0.88	0.88
none	35	2.4	0.85	0.85
12GB	**12**	**3918**	**68.00**	**148.22**
12GB	13	268.5	1.00	2.11
12GB	14	17.8	0.36	0.76
12GB	22	2.4	0.33	0.69

Table 7: Heuristic Performance

Heuristic	Mean Resp	Hit Rate	Modem Frac	Mean Sat	Mean Modem	Modem Queue
Greedy	1.72	0.36	0.517	2.86	2.65	1.33
Sorted	2.29	0.47	0.530	N/A	4.31	1.88
GPop	1.33	0.45	0.415	1.01	2.88	1.45

the higher bandwidth, and shorter queue, in the 35 Mb/s system, cache misses should see lower transfer times in the high bandwidth setting.

The explanation for this seeming discrepancy is that transfer time comprises only a small fraction of response time in high bandwidth environments[13]. OpCaching reduces overhead costs, such as the time to send a request, by reducing contention on the modem uplink. It is for this reason that the lower bandwidth system with OpCaching attains lower response times on cache misses than the higher bandwidth system.

4.4 Heuristic Evaluation

Herein, we compare the performance of the proposed heuristic schedulers using the environment described in Tables 4 and 5. Hereafter, we use 4GB caches at the client nodes. Table 7 summarizes the results. The "Modem Frac" column reports the fraction of requests which are served by modem. "Mean Sat" and "Mean Modem" give the response times for requests served via satellite and modem respectively. "Modem Queue" gives the average size of the link queue seen by requests served via modem.

We see that GPop outperforms both Greedy and Sorted. GPop wins over Greedy for two reasons. First, by employing popularity in its decision, GPop achieves a higher hit rate. Second, Greedy adds objects to the broadcast link queue so long as broadcast is faster than unicast. Thus, when user demand is greater than broadcast capacity, Greedy effectively slows the broadcast link to the same speed as the unicast links. Consequently, as seen in the mean satellite response times, requests serviced via satellite with Greedy do poorly compared to their counterparts under GPop.

We also find that the Sorted scheduler performs quite poorly. It is outperformed by both GPop, and by Greedy, which uses no popularity information at all. At first glance, this is surprising, given that Sorted has the highest hit rate of the three schedulers. To understand Sorted's performance relative to GPop, we examine the fraction of requests satisfied via modems. We observe that the Sorted scheduler has a significantly larger fraction of requests serviced via the modem link than GPop. The reason that Sorted serves a larger fraction of requests via modem is that, whereas GPop (and Greedy) can service a cache miss via either the satellite link or the modem link, Sorted only services misses via the modem link.

To explain Sorted's performance compared to Greedy, we consider the length of modem queues under both schedulers. As noted above, Sorted services all cache misses via modem. This yields longer modem

[13]For example, in the 14 Mb/s case, the expected transfer time for a 7KB object is 71 ms, which is only 10% of the total response time.

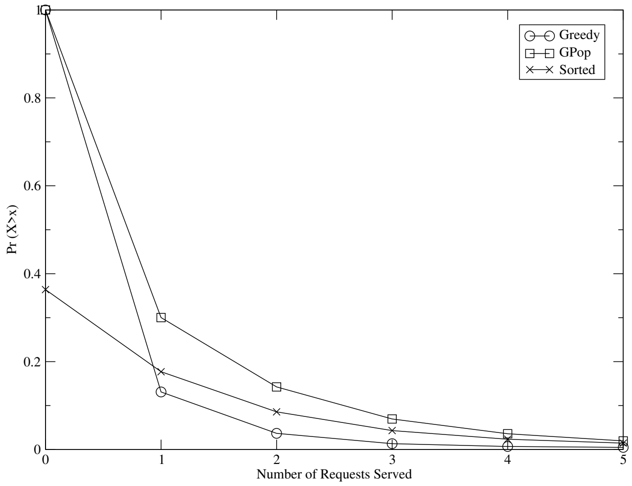

Figure 2: Broadcast Object Utility

queues, and hence, longer response times, for Sorted. Thus, although Sorted serves only slightly more requests via modem than Greedy, it performs worse overall[14].

Sorted clearly uses the links less efficiently than the other schedulers. To understand this inefficiency, we consider the utility of the objects sent over the broadcast link under the three schedulers. Specifically, we ask how many requests are served by each object transmitted over the broadcast link. For the Sorted scheduler, this is exactly the number of cache hits for the object. For the Greedy and GPop schedulers, this is the number of cache hits plus one, as the initial transmission of an object satisfies an outstanding user request.

Fig. 2 plots the inverse CDF of object utility. From this graph, we see that fewer than 40% of the objects that are sent over the satellite with the Sorted scheduler are ever used. This suggests an important counterpoint to Sec. 4.3. While OpCaching is beneficial, using the broadcast channel for *prefetching only* hurts performance because many of the pushed objects are never used.

4.5 Performance Under Varied Loads

Thus far, we have focused on performance under high loads. We now consider performance under fixed bandwidth, but with varied loads. We fix the bandwidth to be 2.25 Mb/sec and scale load by adjusting the mean request rate. This enables us to assess performance under both low and high load conditions. We compare against a simple system ("SatOnly") that employs OpCaching but does not use the modem downlink channels. An ideal system will perform as well as SatOnly under light loads, where the satellite link alone has sufficient capacity, while exhibiting better performance under high loads, where the satellite becomes overloaded. We include ModemOnly, a configuration in which only the modem links are used, for completeness.

In the SatOnly system, as the satellite link becomes overloaded, queue lengths grow to infinity, and response times grow arbitrarily large. In practice, however, users are likely to adapt their behavior to the performance of the system. For example, when the system is sluggish, users may abort their requests. As a simple approximation of this behavior, we implement a per-user limit on the number of outstanding and queued requests. Briefly, a given user may have four requests outstanding (issued to the server) at any given time. When more than four requests are outstanding, further requests are queued. The queue depth is itself limited to four requests. When the queue fills, any further requests generated by the user are discarded.

Fig. 3 presents the fraction of completed requests for Greedy, GPop, Sorted, SatOnly, and ModemOnly. We find that the capacity of the SatOnly system is 0.07 requests/client/sec. Beyond this point, SatOnly fails

[14] In fact, recalling that Greedy effectively slows the broadcast link to the same speed as a modem, the fraction of requests that sees "modem-like" speeds is higher for Greedy than Sorted. This is outweighed, however, by the effect of the long modem queues under Sorted.

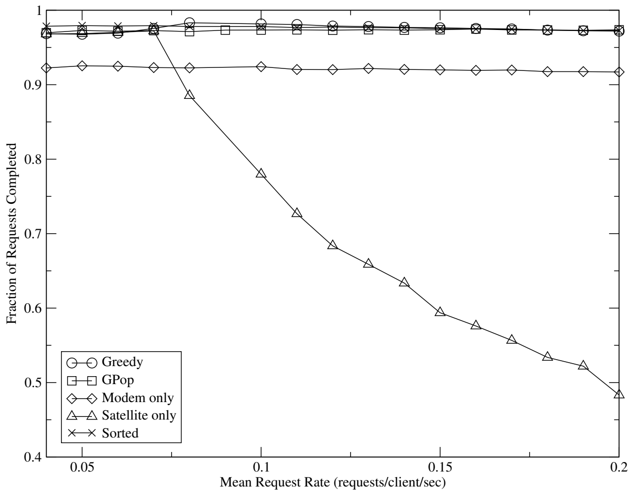

Figure 3: System Capacity

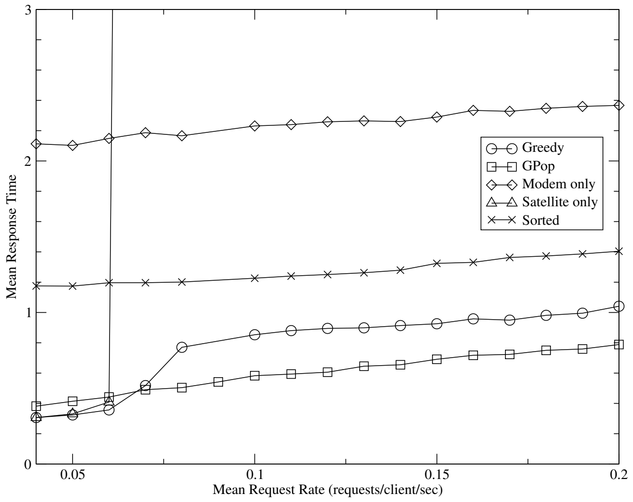

Figure 4: Response Time

to complete a significant fraction of requests. When the system is configured to use modems, in contrast, the system maintains a stable request completion rate. This is as expected, because the modems alone have sufficient capacity to meet user demand.

Fig. 4 presents mean response time for the same scenarios. The most dramatic feature of response time performance is that SatOnly is essentially unusable beyond an offered load of 0.06 requests/client/sec. We note that both GPop and Greedy behave in the manner that we would like. At low loads, they perform comparably to SatOnly. As load increases, they gradually offload requests to the modems (Fig. 5), and their performance degrades gracefully. As explained in Sec. 4.4, GPop outperforms Greedy at high loads. Meanwhile, Sorted performs relatively poorly, due in part to the fact that it serves nearly half of requests via modem links, even at low loads.

4.6 Large Object Performance

Subscribers to high-speed Internet services, are unlikely to be satisfied with fast loading of web pages alone. A large draw of broadband Internet access is the ability to quickly access large files such as music or movie trailers. A danger of focusing on mean response time is that we may neglect the performance of retrieving

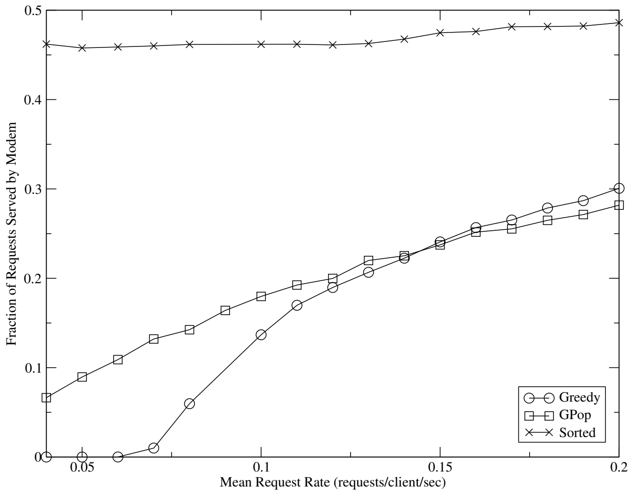

Figure 5: Object Offloading

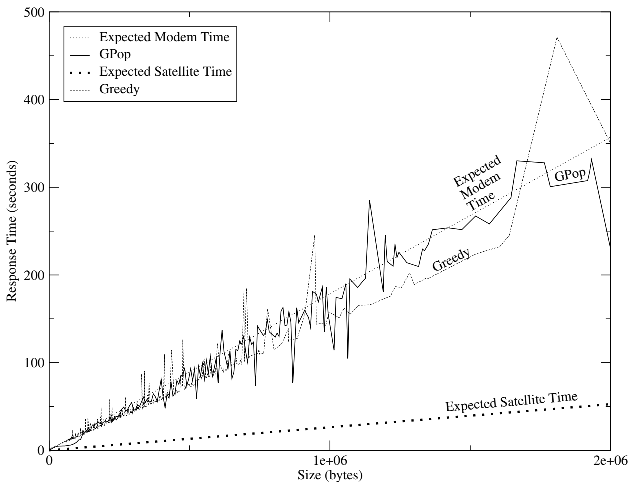

Figure 6: Large Object Performance

these large objects. Specifically, in typical web workloads, large objects account for only a small fraction of the number of requests [15]. Thus, it may be possible to achieve good overall performance, as measured by mean response time, while doing poorly for large objects.

We begin with the best overall performer, GPop. We examine GPop's performance on large objects in Fig. 6. The graph shows response time performance for the configuration described in Tables 4 and 5. "Expected Modem Time" gives the expected service time over a modem link, accounting for contention from other requests. "GPop" gives the actual performance of GPop in simulation. "Expected Satellite Time" is analogous to Expected Modem Time. Ideally, the slope of the GPop line would be close to the slope of Expected Satellite Time. The conclusion that we draw from Fig. 6 is that large requests do in fact suffer poor performance with the GPop scheduler.

To explain why large requests do poorly, we review the GPop algorithm. As explained in Sec. 4.1, GPop serves a request using the satellite link $time_{bcast} \leq time_{ucast} \times satpref \times r$. When $satpref \leq 1$, as in our simulations, an object must have seen multiple requests, or the expected service time on the satellite link must be lower than the expected modem service, for the request to be served via satellite. In particular, with a low $satpref$, an object must be very popular, or the expected satellite service time much lower than the expected modem service time. The latter is infrequently true because the shared nature of the broadcast

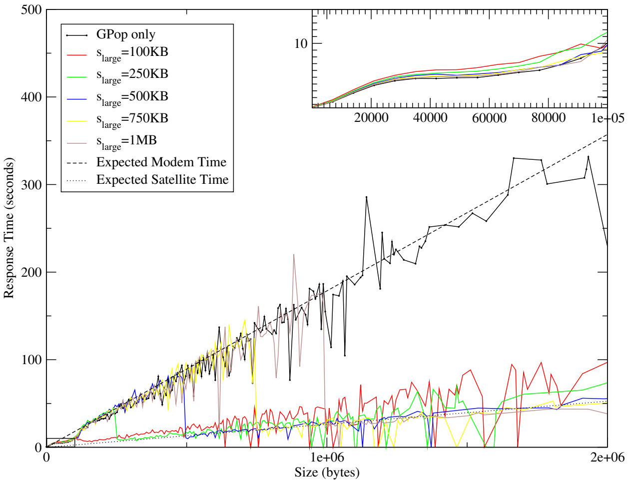

Figure 7: Large Object Performance With GPopLarge. The inset graph focuses on small objects.

Table 8: Overall Performance with GPopLarge

Scheduler	Mean Resp	Std Dev
GPop	1.31s	3.40
s_{large}=100KB	1.72	2.74
s_{large}=250KB	1.56	2.58
s_{large}=500KB	1.46	2.70
s_{large}=750KB	1.42	2.96
s_{large}=1000KB	1.38	3.00

link. Thus, large objects, which are generally unpopular [12], are unlikely to be sent on the satellite link.

We might expect that Greedy, which does not bias against large objects in the manner of GPop, would provide lower response times than GPop on these objects. In Fig. 6 we see that Greedy does do slightly better than GPop, but that the speed of transfer for large objects remains much closer to modem speeds than satellite speeds. This is a direct consequence of Greedy's tendency to maintain long queues on the satellite link, thereby slowing satellite-served requests.

To improve large object performance, we consider a simple revision of GPop, which we call *GPopLarge*. We set a threshold size s_{large} for large objects. Objects larger than s_{large} are sent via the satellite link if they would have been sent via satellite under GPop, or if the expected service time via satellite is less than the expected service time via modem. Given the typical satellite queue lengths in our configuration, the latter is the typical case.

In Fig. 7 we see that this simple approach can significantly improve performance for large objects. A 3MB file, for example, achieves a speedup of 3.79 with s_{large}=100KB as compared with GPop. For s_{large}=500KB, the speedup increases to 6. Thus, even under high loads, large objects can achieve reasonable performance with GPopLarge.

The inset in Fig. 7 shows that the improvements come at the cost of slightly worse performance for small objects. Table 8 quantifies the impact of GPopLarge on mean performance. For s_{large}=100KB and s_{large}=500KB mean response time increases by 31% and 11% respectively. Although the gains in large object performance are not free, they are likely to outweigh the losses. In plain terms, a user is less likely to be concerned about small differences in page load times for small web pages than the ability to access large files, such as music and video, quickly.

The ability to accelerate the retrieval of large objects necessarily depends on the workload. At the extreme, if the workload consists of nothing but requests for large and unpopular documents, then accelerating large objects will be infeasible. For this reason, we evaluate the performance for large objects on heavier

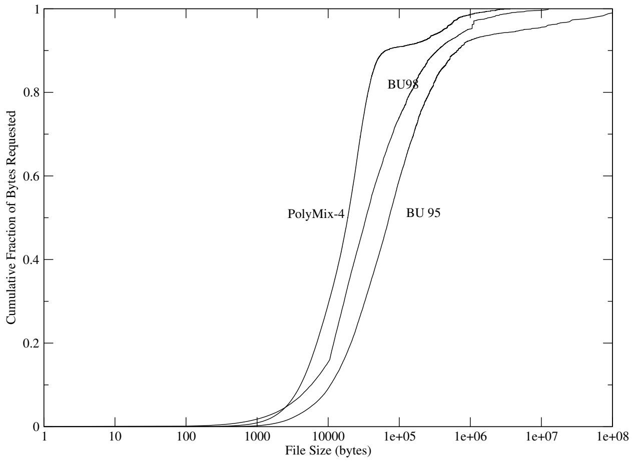

Figure 8: Large Object Load

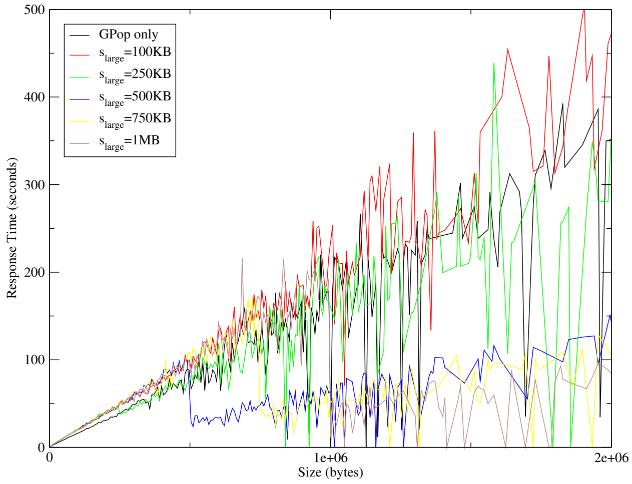

Figure 9: Large Object Performance on BU 95

tailed workloads than PolyMix-4. For this purpose, we configure Polygraph to generate workloads similar to measured web client behavior. Specifically, we configure Polygraph to generate file sizes according to the hybrid Lognormal-Pareto models reported in [15]. Fig. 8 shows the load attributable to large objects in the resulting workloads. The models, based on traces of web users at the Boston University Computer Science Department, clearly show heavier tails than PolyMix-4.

Fig. 9 and Table 9 give results for BU 95 while Fig. 10 and Table 10 give results for BU98. For both BU 95 and BU 98, we find that setting $s_{large} < 500KB$ achieves little improvement. For BU 95, $s_{large} = 500KB$ improves large object response time by a factor of 2.8, with a 20% slowdown on overall mean response time. GPopLarge performs better on BU98, where it achieves a factor 8 speedup for large files, with only a 4.3% slowdown on overall mean response time.

Thus, although the ability to accelerate access to large objects is workload-dependent, GPopLarge achieves significant improvement for typical workloads.

5 Related Work

There is a large body of work on optimizing data retrieval in networks with broadcast or multicast capabilities. We describe the most relevant theoretical and system design work below.

Table 9: Overall Performance on BU95

Scheduler	Mean Resp	Std Dev
GPop	2.02s	15.26
s_{large}=100KB	2.80	18.92
s_{large}=250KB	2.77	33.47
s_{large}=500KB	2.47	9.93
s_{large}=750KB	2.53	8.37
s_{large}=1000KB	2.47	8.36

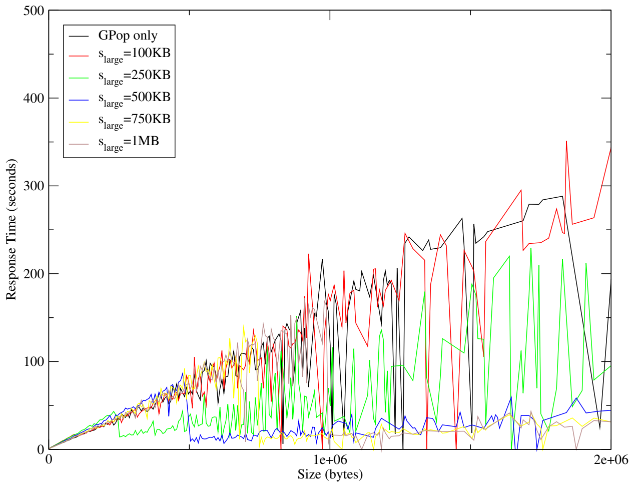

Figure 10: Large Object Performance on BU 98

Table 10: Overall Performance on BU98

Scheduler	Mean Resp	Std Dev
GPop	0.92s	6.78
s_{large}=100KB	1.13	4.73
s_{large}=250KB	1.05	3.38
s_{large}=500KB	0.96	2.99
s_{large}=750KB	0.92	3.19
s_{large}=1000KB	0.92	3.22

Many different variants of the broadcast scheduling problem have been theoretically evaluated in the past. An early study of broadcast scheduling by Acharya and Muthukrishnan [16] provides a nice introduction to the problem. Finding a broadcast schedule which minimizes the maximum response time is considered in [17]. Recently, the problem of minimizing the total response time has been shown to be NP-Hard, even if all the files have unit sizes [18]. This result is interesting as there are very few hardness results for scheduling problems where jobs have the same size. All current results to minimize total response time are limited to using resource augmentation [6] and unit size jobs. Kalyanasundaram et al. [19] give a $\frac{1}{\epsilon}$ speed, $\frac{1}{1-2\epsilon}$ approximation for this problem and Gandhi et. al. [20] give an improved $\frac{1}{\epsilon}$ speed, $\frac{1}{1-\epsilon}$ approximation algorithm.

In Sec. 2.2, we showed that for the case when the file sizes are variable and they have release times, our problem is a more general case of the classic and hard problem of minimizing the weighted response time on a single machine. Becchetti et al. [21] give a $(1+\epsilon)$-speed, $(1+\frac{1}{\epsilon})$ competitive algorithm for the problem of weighted response. Hence, our result (with the same performance guarantee) generalizes their result.

Some past studies consider scenarios closer to our own. For example, [22] shows that there is no $o(\sqrt{n})$ competitive algorithm for sending variable sized objects over the broadcast channel, with no caching. [23] considers a system in which the clients have a cache. However, their focus is on the various memory management policies for the cache and they do not consider the scheduling of the broadcast channel at all.

In summary, while this theoretical problem has received much attention recently, past efforts differ in various fundamental ways from our approach. First, none of the previous works deal with scenarios where the clients have a cache. Secondly, none of them looks at the general problem where there are both unicast and broadcast channels. Thirdly, all of them assume that files have a fixed unit size. These differences play an important role. For example, while the no-cache version is NP-hard even for the unit file size case, our problem can be solved exactly in polynomial time in the case of unit size files.

Several system designs also consider the use of broadcast or multicast to deliver files. [24] and [25] explore a design in which a sender periodically transmits objects from a fixed library across a broadcast channel. In this system, the periodicity of a particular object is governed by its popularity. Other works extend this basic design by incorporating client "pull" requests for objects [26, 27, 16, 28], object dependence relationships (to model embedded objects in web pages) [29] and the use of multiple downlink channels [30, 31].

A few systems explicitly consider the use of satellite channels. [32] proposes an architecture for employing satellites to distribute web content to caches. DirecPC service includes a "webcast" [33] feature, which automatically broadcasts popular websites to users. However, technical details about this service are unavailable.

The prior work in system design differs from our own in three significant ways. First, previous work either considers only a single downlink, or assumes a small number of unicast downlink channels of the same speed as the broadcast channel. Second, most prior work ignores the ability to cache objects requested by other users. Third, we consider a web-like workload of millions of data items while prior work considers smaller workloads of at most ten thousand objects.

6 Conclusions and Future Work

We have considered the problem of optimizing response time in a network comprised of broadcast and unicast links, where caching is inexpensive. A driving force behind our approach is the idea that we can improve performance by exploiting the fundamental properties of system elements.

From a theoretical perspective, we find that both the general problem, and several simplifications, are NP-Hard. Where possible, we give approximation algorithms. Despite the hardness of the problem, we find that heuristic approaches can effectively leverage system elements to improve performance. They provide similar performance to existing systems at low loads, and dramatic improvements in capacity and response time at high loads.

An interesting area of future work is to consider where else basic system properties can be exploited. One example is that broadcast networks enable lightweight invalidation-based consistency management. Without broadcast capability, providing such consistency requires tracking all the users who hold a data item. We need to know both *if* we need to send an invalidation, and *where* to send it. With broadcast, we need only determine *if* an invalidation is required. A mechanism such as leasing [34] might provide a lightweight answer to this question.

7 Acknowledgements

We thank Sanjay Rao, Bianca Schroeder, and Peter Steenkiste for comments on earlier versions of this paper.

References

[1] Hughes Network Systems, "DirecPC," `http://www.direcpc.com`.

[2] Starband Communications, "Starband," `http://www.starband.com`.

[3] DSLReports.com, "Satellite forum FAQ," `http://www.dslreports.com/faq/satellite/all\#2058`, July 2002.

[4] Hughes Network Systems, "Directv: TiVo," `http://www.directv.com/DTVAPP/imagine/TIVO.jsp`.

[5] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker, "Web caching and zipf-like distributions: Evidence and implications," in *INFOCOM (1)*, 1999, pp. 126–134.

[6] Bala Kalyanasundaram and Kirk Pruhs, "Speed is as powerful as clairvoyance," *Journal of the ACM*, vol. 47, no. 4, pp. 617–643, 2000.

[7] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization*, Prentice-Hall, Englewood Cliffs, NJ, 1982.

[8] Michael R. Garey and David S. Johnson, *Computers and Intractability*, W.H. Freeman and Company, N.Y., 1979.

[9] Eugene L. Lawler, "A pseudo-polynomial algorithm for sequencing jobs to minimize total tardiness," *Annals of Discrete Mathematics*, vol. 1, pp. 331–342, 1977.

[10] J. Du and J.Y-T. Leung, "Minimizing total tardiness on one machine is NP-Hard," *Mathematics of Operations Research*, vol. 15, pp. 483–495, 1990.

[11] "ns-2 Network Simulator," http://www.isi.edu/nsnam/ns/, 2000.

[12] The Measurement Factory, "Polygraph," `http://www.web-polygraph.org`.

[13] Polyteam, "Cache-off," `http://cacheoff.ircache.net`.

[14] Roger Rusch, "Forecast for broadband satellite services – part 2," *SKYBroadband Newsletter*, Feb 15 2002.

[15] Paul Barford, Azer Bestavros, Adam Bradley, and Mark Crovella, "Changes in web client access patterns," *World Wide Web, Special Issue on Characterization and Performance Evaluation*, vol. 2, pp. 15–28, 1999.

[16] S. Acharya and S. Muthukrishnan, "Scheduling on-demand broadcasts: new metrics and algorithms," in *MOBICOM*, Dallas, TX, Oct 1998.

[17] Yair Bartal and S. Muthukrishnan, "Minimizing maximum response time in scheduling broadcasts," in *SODA*, 2000.

[18] Thomas Erlebach and Alexander Hall, "NP-Hardness of broadcast scheduling and inapproximability of single-source unsplittable min-cost flow," in *SODA*, 2002.

[19] Bala Kalyanasundaram, Kirk Pruhs, and Mahendran Velauthapillai, "Scheduling broadcasts in wireless networks," in *European Symposium on Algorithms*, 2000, pp. 290–301.

[20] Rajiv Gandhi, Samir Khuller, Yoo-Ah Kim, and Yung-Chun Wan, "Algorithms for minimizing response time in broadcast scheduling," in *IPCO*, 2002.

[21] Luca Becchetti, Stefano Leonardi, Alberto Marchetti-Spaccamela, and Kirk Pruhs, "Online weighted flow time and deadline scheduling," in *Random-Approx*, 2001, pp. 36–47.

[22] Jeff Edmonds and Kirk Pruhs, "Broadcast scheduling: when fairness is fine," in *SODA*, 2002.

[23] Leandros Tassiulas and Chi-Jiun Su, "Optimal memory management strategies for a mobile user in a broadcast data delivery system," *IEEE Journal on Selected Areas in Communications*, vol. 15, no. 7, September 1997.

[24] Stanley Zdonik, Michael Franklin, Rafael Alonso, and Swarup Acharya, "Are "Disks in the Air" just pie in the sky?," in *IEEE Workshop on Mobile Computing Systems and Applications*, Santa Cruz, California, Dec. 1994, pp. 12–19.

[25] Swarup Acharya, Rafael Alonso, Michael Franklin, and Stanley Zdonik, "Broadcast disks: data management for asymmetric communication environments," in *Proceedings of SIGMOD 95*, San Jose, CA, May 1995, ACM, pp. 199–210.

[26] Swarup Acharya, Michael Franklin, and Stanley Zdonik, "Balancing push and pull for data broadcast," in *Proceedings of SIGMOD 97*, Tucson, AZ, May 1997.

[27] Ping Xuan, Subhabrata Sen, Oscar Gonzalez, Jesus Fernandez, and Krithi Ramamritham, "Broadcast on demand: efficient and timely dissemination of data in mobile environments," in *IEEE Real-Time Technology and Applications Symposium*, Montreal, Canada, June 1997, pp. 38–48.

[28] Demet Aksoy and Michael J. Franklin, "Scheduling for large-scale on-demand broadcasting," in *Proceedings of IEEE INFOCOM'98*, San Francisco, CA, March 1998.

[29] Vincenzo Liberatore, "Multicast scheduling for list requests," in *Proceedings of IEEE INFOCOM'02*, New York, NY, June 2002.

[30] Qinglong Hu, Dik Lun Lee, and Wang-Chien Lee, "Performance evaluation of a wireless hierarchical data dissemination system," in *MOBICOM*, Seattle, WA, August 1999, pp. 163–173.

[31] Konstantinos Stathatos, Nick Roussopoulos, and John S. Baras, "Adaptive data broadcast in hybrid networks," in *23rd International Conference on Very Large Data Bases*, Athens, Grece, Sept 1997, pp. 326–335.

[32] Hua Chen, Marc Abrams, Tommy Johnson, Anup Mathur, Ibraz Anwar, and John Stevenson, "Wormhole caching with HTTP PUSH method for a satellite-based web content multicast and replication system," in *Proceedings of the 4th International Web Caching Workshop*, San Diego, CA, March 1999.

[33] Hughes Network Systems, "Introducing webcast," `http://www2.direcpc.com/helpfiles/webcast/webcast_overview.htm`, 2000.

[34] Venkata Duvvuri, Prashant Shenoy, and Renu Tewari, "Adaptive leases: A strong consistency mechanism for the world wide web," in *INFOCOM00*, Tel Aviv, Israel, Mar 2000, IEEE, pp. 834–843.

Table 11: Effect of *satpref* on GPop Performance

satpref	Mean Resp	Sat Util	Mean Sat Queue	Mean Mod Queue
0.02	1.4519	0.899	1.9493	1.501
0.04	1.3804	0.992	3.7719	1.478
0.06	1.3149	0.998	5.8904	1.433
0.08	1.3191	0.999	7.8599	1.423
0.1	1.3089	0.999	10.469	1.405
0.2	1.3254	0.999	20.772	1.375

APPENDIX

Tuning the GPop Scheduler

How should we choose a value for $satpref$? Like Greedy, GPop serves a request over the link that has the higher available bandwidth at the time the request arrives. However, GPop modulates this decision by the popularity of an object, as measured by prior accesses to the object. Considering the case of $satpref = 1$, for example, if object o has three accesses, then GPop will send o over the broadcast link even if the broadcast link has only 1/3 the available bandwidth as the unicast link. Higher values of $satpref$ make GPop more likely to serve a request via broadcast. That is, as $satpref$ increases, the speedup that GPop will require for using the satellite link decreases.

To find a good value for $satpref$, we run simulations with the parameters listed in Tables 4 and 5. We summarize the results in Table 11. We find that the key tradeoff in choosing $satpref$ is balancing satellite and modem performance. As we increase $satpref$, the length of modem queues decrease, and thus, modem performance improves. Simultaneously, satellite queues increase, causing satellite performance to degrade. A good value for $satpref$ balances this tradeoff to optimize response time.

Note that an additional factor should be considered when the GPopLarge scheduler (Sec. 4.6) is used. Because the satellite link is shared fairly between all queued requests, the ability to accelerate large objects will depend on the queue length. More precisely, the acceleration achieved is inversely proportional to the queue length. Hence, $satpref$ should be chosen so as to minimize satellite queue length while still balancing the above tradeoff.