Predictive Caching for Video on Demand CDNs

Bogdan Carbunar
Computing and Information Sciences
Florida International University
Miami, FL
Email: carbunar@gmail.com

Michael Pearce
Motorola Solutions
Schaumburg, IL
Email: michael.pearce@motorola.com

Venu Vasudevan, Michael Needham
Applied Research Center
Motorola Mobility
Schaumburg, IL
Email: {cvv012,cmn002}@motorola.com

***Abstract*—Video on Demand (VoD) services provide a wide range of content options and enable subscribers to select, retrieve and locally consume desired content. In this work we propose caching solutions to improve the scalability of the content distribution networks (CDNs) of existing VoD architectures. We first investigate metrics relevant to this caching framework and subsequently define goals that should be satisfied by an efficient solution. We propose novel techniques for predicting future values of metrics of interest. We use our prediction mechanisms to define the cost imposed on the system (network and caches) by items that are not cached. We use this cost to develop novel caching and static placement strategies. We validate our solutions using log data collected from Motorola equipment from several Comcast VoD deployments.**

I. Introduction

Video on Demand (VoD) systems allow users to select and view content on demand. The content is stored by the VoD operator at various locations in the network and is transferred upon to users upon demand over a content delivery network (CDN). In this paper we consider CDN architectures built over the cable television (CATV) transport network (see Figure 1). In this architecture, the video service office (VSO) is the main site while several video home offices (VHOs) are secondary sites that store the same content but serve smaller, disjoint regions. Current VoD implementations (including Comcast, Charter and Time Warner) require each VHO to store all the content present at the VSO. While enabling high content availability, this solution has significant scalability issues: VHOs need larger disks, RAMs and flash. However, due to different regional consumption patterns part of the content becomes irrelevant [1].

In this work we relax the content duplication constraint and propose that each VHO site is managed independently – the local storage becomes a cache of the central library. This enables hardware scaling to be done based on local demand. However, due to VHO level misses, this approach introduces a trade-off between the additional traffic imposed on the network links to fetch missed items and the hardware scaling cost. As a first contribution, we identify several relevant metrics and associated goals that need to be achieved in order to make this solution functional. A second contribution consists of devising placement and caching algorithms that attempt to achieve these goals. Our solutions build on novel algorithms for predicting

[1]Data from a Time Warner deployment in San Antonio shows that only 8000 items from a 40000 item library were requested during a 3 day interval.

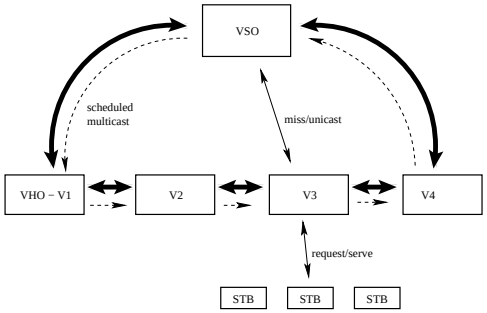

Fig. 1. System Architecture. Thick lines denote the ring topology links, connecting the VSO and the VHOs. Links are bi-directional.

a *penalty* value for each item: the cost of not storing the item for a future interval. In addition, we take advantage of the existing streaming servers at all VHO and VSO sites and use the penalty values of items to drive not only the replacement algorithm (which items to evict from a cache) but also the decision of which items to transfer reliably and cache and which to stream (and not cache).

The caching problem has been studied in a variety of contexts, e.g., Web caching, memory and distributed storage. The seminal work of Dahlin et al. [1] introduced the concept of collaborative caching along with several caching algorithms. Our work differs in that clients can access other caches but cannot decide their membership. Caching for streaming data includes work on prefix caching [2], [3], segment-based caching [4] and multicast cache [5] where the main concern is minimizing the start-up latency. Caching for content distribution networks has also been addressed in a theoretical framework in recent work [6], [7]. Our work differs in that (i) it first identifies the constraints that define the existing CDNs of VoD providers including Comcast, Charter and TimeWarner, (ii) then proposes novel predictive caching solutions and (iii) finally validates the solutions on data collected from Motorola equipment from existing VoD deployments.

We have used Java and ns-2 implementations to evaluate our algorithms on log traces from Motorola VHO servers of Comcast VoD deployments. Our conclusions are that our solutions satisfy all our goals: they significantly reduce the total network traffic (half the value of LRU), improve its distribution on the network links (one order of magnitude better than LRU) and reduce the cache overwrite value per day to a fraction (10-20%) of the cache size.

II. System Model

We consider the content distribution networks of several Comcast, Charter and TimeWarner video on demand deployments. The Video Service Office (VSO) is the central data repository. The VSO processes each content item as it enters the system, packages it and stores it in a local *content library*. The VSO also has a high-capacity streaming server that can stream items directly to users. The Video Home Office (VHO) is a smaller replica of the VSO, serving a geographical subregion of the area served by the VSO. Each VHO consists of a storage component (the cache) and a smaller capacity streaming server that serves items stored in the cache upon user request. Each site V (VSO or VHO) has a unique identifier, Id(V). The VSO and the VHOs are connected through a high speed fiber ring.

Each VHO stores only a subset of the items stored at the VSO. Whenever a miss occurs (a user requests an item not stored on the VHO cache), the VHO needs to fetch the item. The VHO can fetch the item from another VHO site or from the VSO. The source site streams the item directly to the user (from its streaming server) or reliably sends it to the requesting VHO who then caches and streams it to the user.

We have several data sets from VoD deployments in US, China and Europe. We focus here on our largest data set, collected from Motorola VHO equipment from a Comcast VoD deployment in Warren (Detroit, MI). The "Warren" data has been collected over 18 full days from August 16, 2008 to September 2, 2008. The total number of items accessed was 12625 for a total of 4.6 million accesses. Each data set consists of two types of data. The *content database* contains metadata of all content items stored on the VSO. Each entry in the content database refers to one item and lists the item's id, size and consumption rate. In all our logs there are two types of content encoding: standard definition, requiring a bit rate of 3.7Mbps and high definition, with a bit rate of 14.4Mbps. The *stream database* contains information about requests from VoD system users. Each entry refers to one user request and contains a unique stream id, the name of the content consumed, the consumption interval and the requesting IP address.

Our work is based on the observed periodicities in the evolution in time of two metrics: RPM, the (total and per item) number of requests received in a minute and the bandwidth required to satisfy all requests. These metrics vary during a day: high values are recorded around midnight, followed by a decrease to the lowest point at around 6am and then a further increase reaching new picks in the evening. However, this consumption pattern repeats every day and moreover, also exhibits weekly consumption patterns (e.g., items tend to be requested least on Thu. and most on Fri. and Sat.).

III. Metrics

Let $\mathcal{V} = \{V_1, .., V_n\}$ be the set of VHOs in the system and let $\mathcal{L}$ be the set of inter-site links in the system. $\mathcal{L}$ includes also the links adjacent to the VSO. The links are full-duplex, thus bidirectional. Let $MISS(V, \Delta T)$ denote the set of items missed on V during time interval ΔT and let Cache(V) denote the set of items stored on site V at a given time. We now investigate the metrics relevant to the caching framework considered in our work.

Definition 3.1: (Traffic Metrics) The Total Miss Traffic (TMT) is the sum of the size of all the items missed on all VHOs over a time interval ΔT: $TMT(\Delta T) = \sum_{i=1}^{n} Size(I)$, $I \in MISS(V, \Delta T)$, $\forall V \in \mathcal{V}$. The Total Link Traffic (TLT) is the sum of the traffic imposed on all the links in the system: $TLT(\Delta T) = \sum_i Traffic(L_i, \Delta T)$, $\forall L_i \in \mathcal{L}$.

Definition 3.2: (Congestion Metrics) The Bottleneck Link Traffic (BLT) is the traffic imposed on the most utilized link in the system. The Minimum Link Traffic (MLT) is the traffic incurred on the least congested link.

Given the above definitions, we now define the goals that should be satisfied by an efficient solution.

Goal 3.1: (Traffic) Reduce TMT. Reduce TLT-TMT.

Goal 3.2: (Balance) Reduce BLT. Balance the traffic across the network links: reduce BLT-MLT.

Goal 3.3: (User Satisfaction) For any item I being watched at time T by a user, let $t(I, T)$ denote the number of bytes of I transferred to the user up to time T and let $c(I, T)$ denote the number of bytes of I consumed by the user up to time T. Then, at any time T, we need to ensure that $t(I, T) \geq c(I, T)$.

Goal 3.4: (Cache Overwrite) Reduce the amount of data written on the cache to a daily value that is a fraction of the cache size.

Our last goal focuses on one important limitation of cache storage technology: the finite number of program-erase (P/E) cycles of the flash technology.

IV. Approach

The cache of each VHO site is organized into two lists (i) one containing items that are currently consumed – the *viewSet* and (ii) one with items that are not consumed but have not yet been evicted – the *stillCached* list. When a request for an item I is received by a VHO V, if I $\in$ Cache(V), V streams the item to the user via its streaming server. If I $\notin$ Cache(V), V needs to forward this request to other sites that may store I, who then serve this request. V has then the option of storing item I locally. Our approach for making this decision is based on penalties. That is, each VHO assigns a penalty value to each item for which it has received a request. The penalty defines the "cost" of not storing the item in the cache. In the following we briefly outline our approach.

Predicting Penalty Values: We define the penalty of an item to be higher for items of larger size, that are requested frequently and are more difficult to fetch. Predicting the penalty of an item depends on the ability to infer the future number of requests likely to be received for the item and also the future cost of fetching an item over a certain link. In the following we describe our approach in making this prediction using M as the generic metric (e.g., RPM). For each item stored in the system, each VHO site records observed values for M for 7 days, sampled once per minute (1440 values per item per day). We use the recorded history of M to compute a preliminary prediction for future values of

M. For each item I and each minute $T \in \Delta T$, we use a weighted average of values of M recorded during the same minute of each day of the previous week to build an *initial* prediction for the value of M for I at a (future) minute T: $M_{init}(I,T) = \sum_{d=1}^{7} M(I, T - 1440 \times d) \times w_d$. Each of the 7 previous days is assigned a weight, w_d, where $\sum_{d=1}^{7} w_d = 1$. The previous day and the same day one week before have larger weights [2] In Section V and VI we propose two different approaches that use this initial prediction to devise more elaborate and efficient predictions.

Replacement and Streaming Decisions: On each VHO cache, the $stillCached$ items are candidates for eviction when a miss occurs. Let $stillCached = \{I_1, .., I_n\}$ and let $S(I_i)$ be the size of item I_i. When a miss occurs for an item I whose size S(I) exceeds the available cache space, the penalties of I and of all the items in $stillCached$ are computed. Let P(I) be the penalty of I and $P(I_i)$ be the penalty of item I_i from $stillCached$. Then, I is stored in the cache only if there exists a "replacement set", a subset $R = \{I_{i_1}, .., I_{i_r}\}$ of $stillCached$ such that

$$\begin{aligned} \sum_{j=1}^{r} S(I_{i_j}) &\geq S(I) \\ csf \times \sum_{j=1}^{r} P(I_{i_j}) &< P(I) \end{aligned} \quad (1)$$

In case it exists, the replacement set is evicted. Otherwise, item I is streamed directly to the user from another site that stores it. csf, the cache stability factor, defines how fast the replacement algorithm reacts to new items. For large csf values the cache tends to be more static , since new items are being stored less frequently. Note that the replacement set needs to have the minimum penalty among all solutions to Equation 1. The 0-1 knapsack problem can be reduced to this problem, making this problem NP-hard. In the implementation we use a greedy heuristic to compute a candidate replacement set.

Item Retrieval: When a miss occurs at a VHO site for an item I we use a classical distributed hash table approach to discover which other sites store I: Each VHO (including the VSO) is responsible for storing index information about a set of items in the system. The distribution of this index information is performed based on hash values of item ids. Then, for any item I, a *pointer* site – whose id is the closest to the item's id – is responsible for maintaining information about which other sites are storing item I.

V. Prediction Based Caching - PBC

We now propose a prediction based caching (PBC) solution that instantiates the approach described in Section IV. We define the penalty of an item to be proportional to the item's size and popularity: $P(I, \Delta T) = S(I) \times Popularity(I, \Delta T)$.

We use the RPM metric introduced in Section II to determine an item's popularity. PBC divides days into 4 epochs, each 6 hour long. At the beginning of each epoch, RPM_{init} for each item I is computed for each minute of the (next) day using the approach described in Section IV. We define RPM_{pred} to be $RPM_{pred}(I) = \sum_{T=T_0}^{T_1} RPM_{init}(I,T)$, where T_0 is the first and T_1 is the last minute of the epoch. Thus, $RPM_{pred}(I)$ denotes the predicted number of requests for item I during the next epoch.

During each epoch, we record the observed values of RPM for each item I. At each minute $T_c \in [T_0, T_1]$ we define the reactive value of an item I, $RPM_{react}(I, T_c)$ to be $RPM_{react}(I, T_c) = \sum_{T=T_0}^{T_c} RPM(I,T)$. $RPM_{react}(I, T_c)$ denotes the total number of requests seen for I since the beginning of the epoch. We use RPM_{pred} and RPM_{react} to define the popularity of an item I at time T_c, as the weighted average of RPM_{pred} and RPM_{react}: $Popularity(I, T_c) = RPM_{react}(I, T_c) \times \beta(T_c) + RPM_{pred} \times (1 - \beta(T_c)$ The weight β is time dependent. While we have considered also a logarithmic time dependency, in our implementation we have used a linear dependency, $\beta(T) = (T - T_1)/(T_2 - T_1)$, which performed slightly better in our experiments. Given a penalty value for each item, PBC uses Equation 1 to decide which items to stream, which to cache and which to evict. Miss item traffic is scheduled as described in Section IV, using a hop-count metric for choosing the source peer.

Static Placement Algorithm - SPA: We now propose a PBC variant, where the value of the csf factor from Equation 1 is set to infinity. The caching algorithm becomes then a static placement algorithm (SPA): the cache membership does not change when misses occur. In the following we briefly describe how SPA (i) chooses the next cache membership, (ii) schedules the placement of items and how it (iii) handles misses. SPA effectively pre-caches items at the beginning of each epoch: it stores the items that have the largest predicted component of their penalty (up to the space allowed by the cache). Once the cache membership is decided, SPA schedules for placement only the items that are not already in the cache, as follows. The VSO computes the global penalty of each scheduled item I to be $GP(I) = \sum_{i=1}^{n} P(V_i, I)$, where $P(V_i, I)$ is the penalty of I on VHO V_i. The VSO sorts all the scheduled items according to the global penalty and uses a multicast transmission to send them in this order. Upon receiving such a transmission, each VHO uses Equation 1 to decide whether it wants to store the item. Since the static placement traffic occurs in parallel with miss traffic, we set epochs to be one day long and run the static placement algorithm at 6am - the time with the lowest user activity (see Section II). For the remainder of the epoch, missed items are streamed (using the approach described in Section IV) from the closest peer sites storing them.

VI. Network Aware Caching (NAC)

We now propose a caching algorithm that (i) attempts to be more reactive to changes in item popularity and that (ii) takes into consideration the network topology by making the penalty of a (missing) item dependent on the complexity of the fetching process. We call this solution Network Aware Caching (NAC). Given an item I and a future interval ΔT, let $Reqs(V, I, \Delta T)$ denote the number of requests to be received

[2]For instance, for a Sunday, the values of M on Saturday and on the previous Sunday are the most valuable.

for I during interval ΔT at VHO site V. Let $FC(V, I, \Delta T)$ denote the fetch cost of item I for site V.

Definition 6.1: (Network Penalty) The network penalty of an item I at a site V during a future time interval ΔT is $NP(I, V, \Delta T) = S(I) \times Reqs(V, I, \Delta T) \times FC(V, I, \Delta T)$.

We now describe the prediction process for Reqs and FC.

Feedback Reactive Prediction: Using the notation proposed in Section IV, we devise a new prediction solution for a generic metric M, whose initial prediction for a future minute is defined as in Section IV. Let $[T_0, T_1]$ denote one epoch. Each epoch starts with a short warm-up period, defined as the interval $[T_0, T_w]$. During the warm-up period, we use $\sum_{T=T_0}^{T_w} M_{init}$ to define the predicted value for M. Thus, for the first T_w minutes, the predicted value of each item is constant. At any time, including during warm-up, we also record the actual values M experimented by the system. Following the warm-up period, at any time $T_c \in [T_0, T_w]$, we use the predicted M_{init} and values of M recorded from the beginning of the epoch, to evaluate the quality of the prediction so far. That is, for each item I, we define the accuracy of the prediction for metric M at time T_c to be $Acc_M(I, T_c) = \sum_{T=T_0}^{T_c} M(I, T) / \sum_{T=T_0}^{T_c} M_{init}(I, T)$. $Acc_M(I, T_c)$ is computed once per minute following the warm-up period. We scale the initial prediction M_{init} for a future minute T, using the accuracy Acc detected so far: At time T_c, our prediction for the value of M at a future minute T, denoted $M_x(I, T)$, is defined as

$$M_x(I, T) = M_{init}(I, T) \times Acc_M(I, T_c) \quad (2)$$

Acc is computed at time T_c, whereas M_x is the prediction for M at a future time $T > T_c$. Acc is reset at the beginning of each epoch. For NAC, we define the length of an epoch to be one day and the warm-up period to be 30 minutes long. In our experiments, these values have performed best.

Predicting Future Values of Reqs: When a miss occurs at time T_c, $Reqs(I, \Delta T)$ needs to be evaluated for each item I in the cache for the future interval $\Delta T = [T_c, T_c + \delta]$, where δ is a system parameter. We use the RPM metric defined in Section II to define Reqs as the sum of the predicted values of RPM for the interval ΔT: $Reqs(I, \Delta T) = \sum_{T=T_c}^{T_c+\delta} RPM_x(I, T)$ Note that the predicted value $RPM_x(I, T)$ is computed according to Equation 2 defined above.

Defining the Fetch Cost: FC defines the load on the network links when transferring an item I to a site V. As detailed in Section IV, site V first discovers which other sites store item I. It then uses information about those sites, to define $FC(V, I)$ to be the minimum of the cost of all the paths from a site storing I to site V: if $PC(V_i, V_j)$ is the cost of a path between sites V_i and V_j, $FC(V, I) = min\ \{PC(V_j, V, I) | \forall V_j \in \mathcal{V}\ s.t.\ I \in Cache(V_j)\}$. We define the cost of a path for an item I to be the time to transfer I over that path, which is the time to transfer the item over the bottleneck link of the path: $PC(V_i, V_j, I) = max\ \{TransferT(l, I) | \forall l \in Path(V_i, V_j)\}$, where TransferT(l,I) defines the time to transfer I over a link l.

To compute TransferT, we first define a new metric. For any link l, let FPM be the number of flows per minute that traverse l. We use Equation 2 to compute future values of FPM. Each site records FPM values for all adjacent links. Then, a token based approach is employed to collect relevant values (more details in the journal version). Given FPM, TransferT is computed iteratively: At time T_c, compute the prediction $FPM_x(l, T_c + 1)$ and use it to predict how many bytes can be sent during minute $T_c + 1$ over link l (BPM, Bytes Per Minute), using the formula $BPM_x(l, T) = Cap(l)/(FPM_x(l, T) + 1)$. Continue this process, computing $BPM_x(I, T_c + 2)$,.., $BPM_x(I, T_c + T_f)$, until the sum of all BPM values, $\sum_{T=T_c}^{T_f} BPM(l, T)$ exceeds or equals Size(I). Then, set $TransferT(I, l) = T_f$.

VII. Evaluation

We have conducted our evaluation using a combination of Java and ns-2. We have compared the performance of PBC, SPA, NAC and LRU using log data collected from Motorola equipment deployed in the Comcast VoD deployment described in Section II. The system consists of 4 VHO sites and one VSO (see Figure 1), connected by a 1Gbps full-duplex fiber ring. We set csf to 10 for all PBC runs.

A. VHO Level Measurements

We focus first on the performance exhibited by the 4 tested algorithms at one VHO. Figure 2 shows our results for VHO V_1 of Figure 1. Figure 2(a) shows that SPA consistently exhibits the highest miss rate and on average LRU has the lowest miss rate. Figure 2(b) shows that the cache overwrite values of PBC, NAC and SPA are by far smaller than that imposed by LRU on the cache. NAC and PBC overwrite at most 380 GB per day, whereas LRU overwrites up to 5.3 TB (more than the cache size). Figure 2(c) shows the per day TMT. SPA generates the most traffic, almost 10 TB per day. NAC and PBC frequently reduce the TMT at half the values imposed by LRU or SPA.

B. Traffic Load

Figure 3(a) shows the total link traffic (TLT) per day imposed by each tested algorithm. For most days PBC and NAC reduce the TLT to half of LRU or SPA. Figure 3(b) shows the traffic imposed on the most congested link. The traffic on the bottleneck link is significantly lower for PBC and NAC when compared against LRU (between 1.9-4.6TB for PBC and 7.1-10.8 TB for LRU). SPA achieves a maximum bottleneck link traffic of almost 20TB per day. Figure 3(c) shows the load balance achieved by the three algorithms, with a logarithmic y axis. PBC and NAC's balance is one order of magnitude better than that of LRU and SPA (hundreds of GBs per day when compared to 10 TB of LRU and 20 TB per day imposed by SPA).

In the following experiment, performed using ns-2, we study the effects of link congestion by measuring the number of flows simultaneously occurring during the 10th day of the Warren data set, as generated by PBC, NAC and LRU. Figure 4(a) shows the evolution of the number of simultaneous

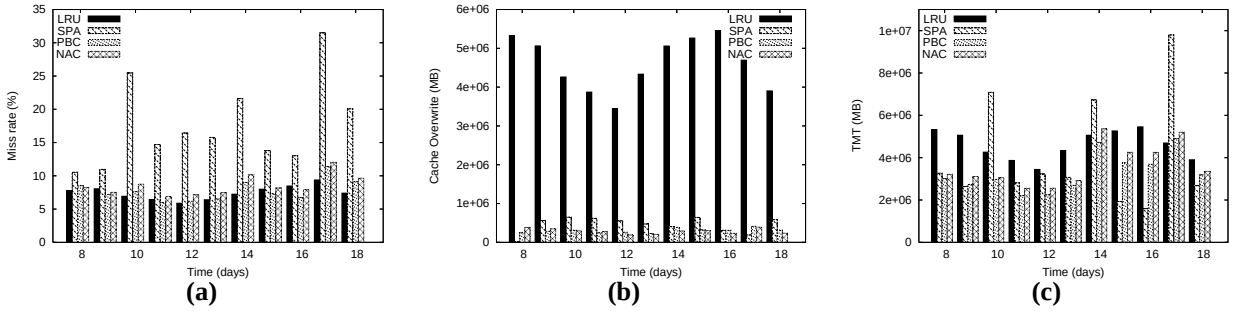

Fig. 2. LRU vs. SPA vs. PBC vs. NAC. (a) Miss rate. (b) Cache Overwrite: PBC, NAC and SPA overwrite the cache an order of magnitude less than LRU. (c) TMT: PBC and NAC generate half the traffic of LRU or SPA.

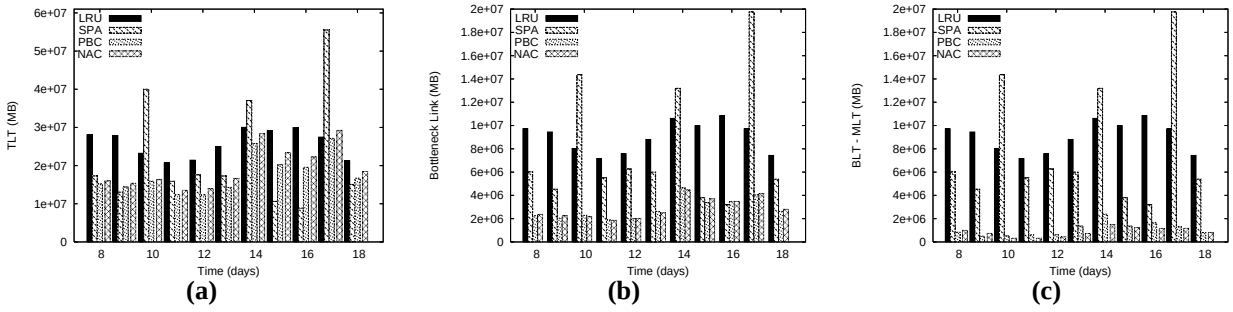

Fig. 3. Load imposed on the network by PBC, NAC, SPA and LRU. (a) Per day TLT. (b) Bottleneck link: PBC and NAC have bottleneck links of much less than half of those of LRU and SPA. (b) Balance: PBC and NAC are one order of magnitude better than LRU and SPA.

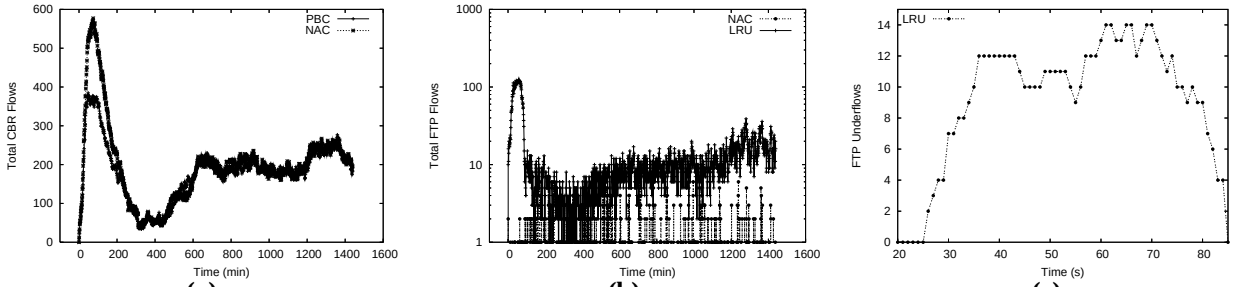

Fig. 4. NS-2 Comparison for PBC, NAC and LRU: (a) Number of simultaneous streams: PBC outperforms NAC. (b) Number of simultaneous reliable transfers (FTP). (b) Number of underflows for LRU: LRU does not support the user satisfaction goal.

streams (CBR flows) imposed by PBC and NAC (1 minute granularity). LRU does not stream. PBC outperforms NAC at the beginning of the day. Later in the day the two algorithms exhibit similar performance, when both are able to make more accurate predictions. Figure 4(b) shows the number of simultaneous FTP flows imposed by NAC and LRU (y axis shown in logarithmic scale). NAC generates mostly streams, with only up to 7 simultaneous FTP flows, an order of magnitude less than LRU. The number of FTP flows imposed by LRU is larger at the beginning of the day. During that time, some of the LRU flows do not perform at the required transmission rate. Figure 4(c) shows the number of underflowing FTP transfers (up to 14 for LRU): flows that cannot achieve a transfer rate exceeding the item's consumption rate.

VIII. Conclusions

In this paper we identify the constraints that define the CDNs of existing VoD providers. We define metrics and goals that should be satisfied by efficient solutions, then propose new caching and placement algorithms. Using data collected from Motorola equipment from existing VoD deployments we show that our solutions satisfy the proposed goals.

References

[1] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. Cooperative caching: using remote client memory to improve file system performance. In *Proceedings of the 1st USENIX OSDI*, 1994.

[2] Subhabrata Sen, Jennifer Rexford, and Don Towsley. Proxy prefix caching for multimedia streams. In *Proceedings of IEEE INFOCOM*, 1999.

[3] Bing Wang, Subhabrata Sen, Micah Adler, and Don Towsley. Optimal proxy cache allocation for efficient streaming media distribution. *IEEE Trans. on Multimedia*, 2004.

[4] Kun-Lung Wu, Philip S. Yu, and Joel L. Wolf. Segment-based proxy caching of multimedia streams. In *Proceedings of WWW*, 2001.

[5] Sridhar Ramesh, Injong Rhee, and Katherine Guo. Multicast with cache (mcache): An adaptive zero-delay video-on-demand service. In *Proceedings of IEEE Infocom*, pages 85–94, 2001.

[6] Sem C. Borst, Varun Gupta, and Anwar Walid. Distributed caching algorithms for content distribution networks. In *Proceedings of IEEE INFOCOM*, 2010.

[7] M.M. Amble, P. Parag, S. Shakkottai, and L. Ying. Content aware caching and traffic management in content distribution networks. In *Proceedings of INFOCOM*, 2011.