Asymmetric Caching: Improved Network Deduplication for Mobile Devices*

Shruti Sanadhya,[1] Raghupathy Sivakumar,[1] Kyu-Han Kim,[2] Paul Congdon,[2]
Sriram Lakshmanan,[1] Jatinder Pal Singh[3]

[1] Georgia Institute of Technology, Altanta, GA, U.S.A.
[2] Hewlett-Packard Laboratories, Palo Alto, CA, U.S.A.
[3] Xerox PARC, Palo Alto, CA, U.S.A.

shruti.sanadhya@cc.gatech.edu, siva@ece.gatech.edu, {kyu-han.kim, paul.congdon}@hp.com,
sriram@ece.gatech.edu, jatinder@stanford.edu

ABSTRACT

Network deduplication (dedup) is an attractive approach to improve network performance for mobile devices. With traditional deduplication, the *dedup source* uses only the portion of the cache at the *dedup destination* that it is aware of. We argue in this work that in a mobile environment, the *dedup destination* (say the mobile) could have accumulated a much larger cache than what the current *dedup source* is aware of. This can occur because of several reasons ranging from the mobile consuming content through heterogeneous wireless technologies, to the mobile moving across different wireless networks.

In this context, we propose *asymmetric caching*, a solution that is overlaid on baseline network deduplication, but which allows the *dedup destination* to selectively feedback appropriate portions of its cache to the *dedup source* with the intent of improving the redundancy elimination efficiency. We show using traffic traces collected from 30 mobile users, that with asymmetric caching, over 89% of the achievable redundancy can be identified and eliminated *even when the dedup source* has less than *one hundredth of the cache size as the dedup destination*. Further, we show that the ratio of bytes saved from transmission at the *dedup source* because of asymmetric caching is over 6× that of the number of bytes sent as feedback. Finally, with a prototype implementation of asymmetric caching on both a Linux laptop and an Android smartphone, we demonstrate that the solution is deployable with reasonable CPU and memory overheads.

Categories and Subject Descriptors

C.2.m [**Computer-communication networks**]: Miscellaneous—*mobile networks, deduplication*

*This work was funded in part by the National Science Foundation under grant CNS-1017234 and the Georgia Tech Broadband Institute.

MobiCom'12, August 22–26, 2012, Istanbul, Turkey.

Keywords

Network deduplication, asymmetric caching, mobile traffic, mobile networks, mobile devices, bandwidth conservation

1. INTRODUCTION

Several recent efforts have established that there are considerable redundancies in network traffic [1, 2, 3] that are not fully leveraged by application layer approaches [4, 5, 6]. Network deduplication (dedup[1]) is a class of solutions that exploits such redundancies to improve network performance [1, 3, 7, 8]. Briefly, a dedup source (`dd-src`) intercepts traffic coming from the sender; segments the traffic into chunks of byte-sequences; and sends across only the hash of a byte-sequence if it is a repeating sequence. The dedup destination (`dd-dst`) then inflates any hashes back to the original byte-sequences and forwards the traffic to the eventual receiver. The reduction in the number of bytes sent between the `dd-src` and `dd-dst` results in improved network performance by allowing the network to sustain a greater traffic load and by reducing congestion levels for a given volume of traffic.

The application of dedup to wireless environments is an attractive proposition due to the ever-prevalent pressures on wireless capacities. Wireless service providers continually attempt to support more users and higher traffic loads without having to use additional spectrum, and dedup is a viable low-cost solution to do so. A straightforward deployment of dedup in a wireless environment would involve the `dd-src` residing in the wireless service provider's network (e.g. the SGSN in 3G) and the `dd-dst` residing on the mobile.

We argue in this paper that dedup faces a unique challenge when used in mobile environments. In static wireline environments, the `dd-src` is typically aware of the complete cache at the `dd-dst` by virtue of being on the data path to the destination. However, there exist several mobile scenarios where the `dd-src` is likely to have knowledge of only a small subset of the cache at the mobile. We elaborate on such scenarios in Section 2. However, if such an asymmetry exists between the caches at the `dd-src` and at the `dd-dst`, the efficiency of traditional dedup techniques is a restricted function of the smaller cache.

In this context, we consider the following question: *How can all of the past cached information at the mobile be successfully leveraged for dedup by any given* `dd-src`*?* In answering the

[1]For brevity we refer to network deduplication as dedup in rest of the paper.

question we categorize traditional dedup techniques as *symmetric caching* techniques where the dd-src and dd-dst maintain identical caches. We then introduce a new approach for dedup in mobile environments called *asymmetric caching*.

Fundamentally, asymmetric caching allows for the cache at the dd-dst to be larger than that at the dd-src. However, it enables the dd-dst to send feedback about portions of its cache to the dd-src. The feedback, sent in real-time, is selected to be pertinent to the ongoing traffic flow. The dd-src thus performs its operations not just based on its regular cache, but also based on the feedback received. The feedback selection problem is non-trivial because not only is the goal to increase redundancy elimination, but also to achieve a high feedback efficiency (ratio of bytes saved from transmission to bytes sent as feedback). Thus, the core of the asymmetric caching solution consists of an *application agnostic mechanism that can partition past and current traffic into contiguous byte sequences called* **flowlets** based on stationarity when the underlying byte stream is considered as a time series. Subsequently, feedback selection occurs by matching an arriving flowlet with a past flowlet, and then choosing content appropriately from the past flowlet to send as feedback. We elaborate on these mechanisms in Section 3.

Asymmetric caching achieves considerably better redundancy elimination by virtue of exploiting a much larger cache at the dd-dst. We show later with trace driven evaluations that asymmetric caching can increase redundancy elimination by over 100%, and provides such improvement even when the cache size at the dd-src is a fraction of that on the dd-dst. Furthermore, asymmetric caching achieves a feedback efficiency (ratio of bytes saved from transmission using feedback to bytes sent as feedback) of over $6X$. In other words, for every byte of feedback sent upstream, 6 $bytes$ of downstream data are saved. In terms of adoption, asymmetric caching can be incrementally and independently deployed. A wireless service provider can thus deploy asymmetric caching to gain from all the past cached content accumulated on the mobile *without requiring any cooperation from other service providers the mobile might utilize*. Also, using prototype implementations of asymmetric caching on a laptop (Linux) and a smartphone (Android), we demonstrate that the CPU and memory overhead are quite reasonable.

In the rest of the paper we introduce the concept of asymmetric caching for dedup in mobile environments. We also answer several questions that arise including the following: How is the feedback chosen to make dedup perform better? Are the dedup benefits with asymmetric caching significant enough to justify the cost of the feedback? How much does asymmetric caching improve performance over a symmetric caching solution in a mobile environment? What are the overheads of implementing asymmetric caching on standard mobile platforms?

2. SCOPE AND MOTIVATION

2.1 Scope

The focus of this work is to enable better wireless network performance through the use of improved network deduplication for mobile devices. The technical contributions of the work broadly apply to a variety of mobile devices and networks. Nevertheless, we restrict the scope of the proposed work as follows:

- We specifically focus on **laptops** and **smartphones** as the mobile devices of interest, and **3G** and **WiFi** as the wireless technologies used at such devices.
- With regard to the wireless environment, since spectrum is inarguably more expensive in 3G environments, we primarily focus on 3G networks as the target environment for the proposed solution, but consider devices that consume content through both 3G and WiFi. However, our proposed solutions can be deployed in WiFi environments as well if required.
- While dedup can be deployed in an end-to-end fashion, we focus on a **last-hop layer 2.5 deployment** model for this work. In such a model, the dedup functionality is realized at entities on either side of the wireless link. While the mobile device is the only candidate deployment location for the dedup-dst, the dedup-src deployment location is likely to be a node such as the Serving GPRS Support Node (SGSN).[2] *In the rest of the paper, for brevity, we generically refer to the upstream dedup node as the* dd-src*, and the downstream dedup node as* dd-dst. Where the deployment location is relevant, we assume the SGSN as the deployment location for the dedup-src. However, the solution may be deployed in possibly other nodes (e.g. a dedicated dedup server) inside the wireless service provider's network.
- Finally, we restrict our focus in this work to dedup on the downstream and on only unencrypted traffic. Wireless traffic remains dominantly downstream and a significant portion of the traffic is not end-to-end encrypted. Hence, we believe that the contributions in the paper will have significant impact in spite of these restrictions. We leave for future work the extensions of the proposed strategies for upstream and end-to-end encrypted traffic.

2.2 Motivational Scenarios

Traditional network deduplication solutions require the dd-src to rely only upon portions of the dd-dst's cache that it is aware of. Such knowledge at the dd-src is implicitly accumulated when the corresponding data traffic flows through the dd-src en-route to the dd-dst. For static wireline hosts, such an arrangement is quite sufficient as the dd-src is always likely to be along the datapath to the dd-dst.

The basic premise of this work, however, is that for mobile devices using wireless connections, the dd-src is likely to be aware of only a fraction of the cache at the dd-dst. Thus, the dd-src is unable to perform deduplication to the fullest extent possible.

We now provide three scenarios in which the above *disconnect between the contents of the* dd-dst *cache and the* dd-src *cache* manifests itself.

- Multi-homed Devices: Most mobile devices today consume content through heterogeneous interfaces. WiFi is the preferred access technology when available due to its low cost and high data rate properties. However, 3G is the access technology used when users are not at locations with WiFi access. Recent studies of wireless data usage have profiled how both technologies are heavily used by mobile devices [9]. Moreover, cellular data offloading to WiFi is observed uniformly across both laptops and smartphones, and across different smartphone platforms [10].

  With traditional dedup, such data access offloaded to WiFi cannot be leveraged for redundancy elimination when the 3G interface is used, because the dd-src is different.

[2]We consider the SGSN as the point of upstream deployment as opposed to the Gateway GPRS Support Node (GGSN) since it already performs per-user functions such as encryption. The dedup-src can be Packet Data Service Node (PDSN) in CDMA networks, or the Access Point for WiFi networks.

- Resource Pooling: Cellular providers have increasingly started to perform IP core resource pooling that is part of the 3GPP standard. SGSN pooling is an example of this trend. In traditional GPRS networks, each SGSN, for example, is wholly responsible for its own service area. However, with SGSN pooling in 3G networks, all the SGSNs in the network work together, and the capacity load between them is distributed by the base station controllers (BSCs) and radio network controllers (RNCs). All BSCs and RNCs are connected to all SGSNs. Any mobile attached to the network is dynamically routed to the SGSN as per the current load distribution [11].

  Thus, the specific SGSN that serves a mobile at a certain point in time does not need to be the same SGSN that serves the mobile at a different time. If traditional dedup were to be used, the `dd-src` at a subsequent SGSN will be unable to use the entire data cache at the mobile because it has no knowledge of the cache entries accumulated through a different SGSN.[3]

- Memory Scalability: With traditional dedup, the `dd-src` dynamically creates a complete date cache for each associated `dd-dst`. A single SGSN typically serves 100,000-1,000,000 simultaneously attached users [12]. Even if SGSN pooling were not to be performed, requiring the SGSN to maintain persistent state across the different attachment sessions for a mobile is thus quite prohibitive. Thus, if the cache state per user is maintained only during the lifetime of that attachment session, then all data accumulated through past sessions will go unused.

All of the above scenarios point to the need for an approach that allows the `dd-src` to leverage the full extent of the cache at the mobile device, even if it might not have prior knowledge of the entire cache. In the rest of this section, we outline any additional goals we want such a solution to satisfy.

2.3 Goals

One approach to address the above-discussed problem is to enable the `dd-src` to fully leverage the cache at the mobile device, and therein increase the dedup efficiency for the downstream communication. However, the following additional goals are critical for the design of such a solution:

- Overall efficiency: While increasing dedup efficiency has the implicit result of helping in spectrum conservation by decreasing network load downstream, any solution has to be explicitly successful in using the overall spectrum (including both upstream and downstream) more efficiently.

- Application agnostic: Network deduplication is a generic technology that is application agnostic. Therefore, any solution to improve dedup has to remain application unaware, and hence applicable to any application used on the mobile device.

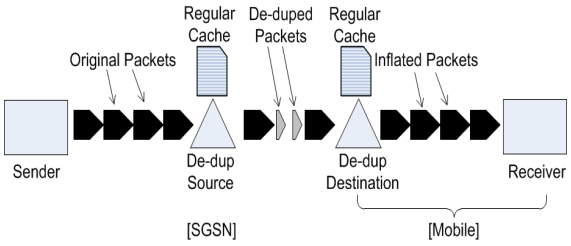
- Limited overheads: Both ends of the dedup solution (the SGSN or WiFi access point upstream, and the mobile device downstream) are resource constrained environments. Hence, any solution to improve dedup performance has to have deployable computational and memory complexities.

[3]SGSN pooling does not involve state transfer between SGSNs.

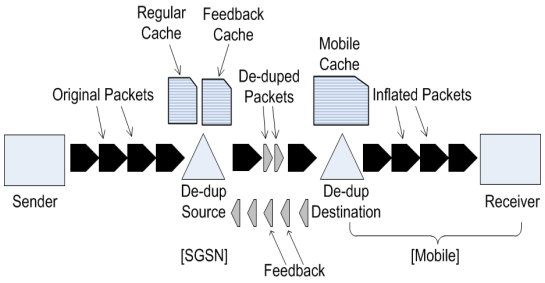
2.4 Background: Baseline Dedup

Network deduplication has been widely studied in related work, and in this paper we use the well known byte-sequence caching as the baseline dedup technique. [1] and [7] first introduced the concept of network deduplication, while [2] and [13] studied characteristics of real network traffic to establish that there indeed exists considerable amounts of redundancy that can be eliminated. More recently, [3] and [14] have proposed techniques to leverage redundancies for traffic reduction in an end-to-end fashion and by overhearing content in wireless networks, respectively. The byte-sequence based dedup is commonly used in both commercial WAN and storage optimization products (e.g., [8, 15]) and related research (e.g., [1, 2, 3, 7, 13, 14]).

Figure 1(a) depicts the operations of the byte-sequence caching. As shown in the figure, the byte-sequence caching algorithm essentially optimizes downlink traffic by replacing previously transmitted byte-sequences or segments of a packet with shorter hashes. Once the base-station receives a downstream packet destined to a mobile, it decomposes the packet into segments using Rabin Fingerprinting [16]. For each of the k segments of a packet, the hashes $[H_1, H_2, ...H_k]$ are computed using a known hashing algorithm such as Jenkins. If any of the hashes H_i is found in the cache, the corresponding segment in the packet is replaced by its hash. The resulting packet that includes both hashes and previously unsent data segments is then transmitted to the mobile. By virtue of the hashes being shorter, the load on the wireless link is reduced. At the mobile, the hashes for the data segments are computed and added to the hash table. Hashed segments are replaced with the corresponding original data before the packet is passed on to higher layers at the mobile.



(a) Baseline dedup (byte-sequence caching)



(b) Asymmetric caching

Figure 1: Baseline dedup vs. Asymmetric caching: Asymmetric caching is an overlay on baseline dedup.

3. ASYMMETRIC CACHING

In the rest of the paper, we present *asymmetric caching*, a solution that satisfies the goals identified earlier. Asymmetric caching performs dedup on the unencrypted downlink network traffic from the `dd-src` (at the SGSN) to the `dd-dst` (at the mobile) as shown

in Figure 1(b). It is built atop a baseline dedup algorithm such as the one described in Section 2.4. At a high level, asymmetric caching enables the dd-dst to send timely feedback to the dd-src about selected portions of its cache. The feedback should be such that the redundancy elimination efficiency, when the dd-src uses both its regular cache and the feedback, approaches that of a scenario where the dd-src has complete knowledge of the dd-dst's cache. While we present the details of the approach in Section 4, we describe the key design elements in the rest of this section.

3.1 When is the feedback sent?

Asymmetric caching uses a *reactive strategy* for sending feedback. Feedback is sent upstream only when data traffic is flowing to the destination. The matching of arriving content with past data in the cache is explicitly used for the selection of feedback that is likely to be most useful. Such a reactive strategy for sending feedback improves the redundancy elimination efficiency in the downstream *while maximizing the feedback efficiency*. An alternative proactive strategy that sends feedback even during idle downstream periods might have a better redundancy elimination performance by virtue of being able to send a larger volume of feedback. However, such an approach will not fare well in terms of feedback efficiency.

3.2 Where from is the feedback chosen?

The dd-dst cache, in asymmetric caching, is partitioned into *flowlets*. Each flowlet is a contiguous subset of a byte stream. The currently arriving traffic is partitioned into flowlets, and any arriving flowlet, say $flowlet_{arr}$ is matched with one of the past flowlets in cache. The feedback is then selected from that past flowlet.

The concept of flowlets is motivated by the fact that most content arriving at the destination is a composition of a collection of objects. For example, an HTTP connection carries different objects such as JPEG images, HTML files, CSS scripts, etc.; an SMB connection carries independent blocks of data as per the scope and sequence of requests; and a peer-to-peer application connection carries different chunks of data, not necessarily contiguous, as requested by the receiver. Thus, a desirable approach for matching arriving traffic to past content would be to match the currently arriving object to an object in the past, and then select feedback from that past object. This would enable the selection of relevant feedback and hence will be favorable to feedback efficiency. While objects may be identified easily if application knowledge is used, such an approach would violate the goal to remain application agnostic. Instead, in asymmetric caching, flowlets are considered as approximations of underlying objects in the data traffic, but are extracted using purely application unaware techniques. We elaborate on the approach next.

3.3 How are flowlets extracted?

In keeping with the application agnostic goal of the design, asymmetric caching employs statistical segmentation to break each downstream flow[4] into *flowlets* without any knowledge of the application. Asymmetric caching relies on the stationarity of the content of individual objects when considered as a time series to perform the segmentation. Thus, changes in the statistical distribution of the underlying byte stream is used to identify flowlet boundaries. Each flowlet is then stored as a sequence of hashes at the dd-dst cache. For any $flowlet_{arr}$, the dd-dst selects hashes from the past flowlet that matches the most.

[4]A flow consists of contiguous bytes/packets with same (source IP, destination IP, source port, destination port) tuple. A connection consists of an upstream and downstream flow.

The flowlet segmentation used in asymmetric caching is a variant of the strategy originally introduced in [17]. The segmentation strategy in [17] segments a piecewise stationary time series $\{X_1, X_2, ..., X_N\}$ into several separate time series $\{X_i, ..., X_j\}$, which are individually stationary. In asymmetric caching, the sequence of bytes in a flow is considered to be a time series, but the algorithm is simplified to grossly approximate position of the boundaries between the time series in return for a lowered computational complexity that is better suited for a resource-constrained mobile environment.

The approach takes a parameter l that is the minimum number of observations required to estimate reliable statistics of a series. At any given location $s(> l)$ in the series, three segments of the series are considered: segment from X_0 to X_s, segment from X_s to X_{s+l} and the aggregated segment from X_0 to X_{s+l}. An autoregressive (AR) model of order p is attempted to be fit on each segment, i.e.

$$X_i = \sum_{j=1}^{j=p} a_i X_{i-j} + \sigma\epsilon \tag{1}$$

on each segment, where ϵ is a white noise (error term).

The gain ($\sigma^2_{a:b}$) of the white noise for the best fitting model on segment $\{X_a, ...X_b\}$ is computed from the sample covariance matrix of that segment. Next, a distance value $d_{0:s:s+l}$ is computed as:

$$d_{0:s:s+l} = (s+l)\log\sigma^2_{0:s+l} - s\log\sigma^2_{0:s} - l\log\sigma^2_{s:s+l} \tag{2}$$

This is intuitively the extra power of the white noise (error) if the two segments are considered in one model as opposed to being in separate AR models. If this distance is more than a given threshold d_{thresh}, a boundary is said to exist between s and $s+l$. If a boundary is not detected, the next considered boundary is after l bytes. An empirical evaluation of the above solution shows that when presented with a mixed-source traffic consisting of different object types such as JPEG, TXT, XML, etc., the object byte boundaries are indeed approximately identified.

Finally, the flowlet in cache that has maximum number of matching hashes with the arriving traffic is identified to be $flowlet_{match}$, the flowlet from where feedback is selected.

3.4 How is the feedback selected?

Once $flowlet_{match}$ is identified, the specific feedback to be sent is selected based on two parameters: the location of the last matching hash between the $flowlet_{arr}$ and $flowlet_{match}$ and the latency for feedback on the upstream. Specifically, the location of the last matching hash in $flowlet_{match}$ offset by δ hashes, where δ is the number of segments that the dd-src is likely to have transmitted before the feedback reaches, is used as the start point for the feedback. δ depends on the average segment size, upstream data rate, and downstream data rate; all parameters computable at the dd-dst. γ hashes are selected from the start point, aggregated into a single packet and transmitted. Note that to ensure that the feedback is always new, the mobile keeps track of all the hashes that have occurred in past downstream packets from this dd-src or have been sent upstream in the past. Such hashes are explicitly removed from any feedback.

3.5 How is the feedback used?

Finally, the dd-src maintains a *feedback cache* in addition to its regular cache. Structurally, the feedback cache is identical to the regular cache and consists of a list of hashes available at the dd-dst. However, the feedback cache is populated only with hashes received through explicit feedback from the dd-dst. When

Algorithm 1 Operations at the dd-src

Input: in_packet = Packet received
Variables:
$regular\_cache$ = Regular cache at dd-src
$feedback\_cache$ = Feedback cache at dd-src
out_packet = Packet to be sent
pkt_chunks = List of chunks
pkt_hashes = List of hashes
seg_hash = Hash of a single chunk
on_flow = Flow to which in_packet belongs
$shim\_hdr$ = 16-bit header: first bit tells if the following sequence is a hash or original text, next 15 bits are used to specify the length of the sequence
Functions:
$hash(chunk)$ = Compute hash of $chunk$
$rabinFingerprints(string)$ = Return value based chunks of $string$
dd-srcDedup(in_packet)

```
1: if in_packet is going to dd-dst then
2:     pkt_chunks ← rabinFingerprints(packet)
3:     for each chunk in pkt_chunks do
4:         if hash(chunk) in regular_cache or hash(chunk) in feedback_cache then
5:             out_packet ← out_packet + shim_hdr + hash(chunk)
6:         else
7:             out_packet ← out_packet + shim_hdr + chunk
8:             add hash(chunk) to regular_cache
9:         end if
10:    end for
11:    send out_packet to dd-dst
12: else[packet is coming from dd-dst]
13:    pkt_hashes ← hashes in in_packet
14:    for each seg_hash in pkt_hashes do
15:        add seg_hash to feedback_cache
16:    end for
17:    out_packet ← IP and TCP headers of in_packet
18:    send out_packet to upstream node
19: end if
```

data arrives at the dd-src, each of its hashes is first looked up in the regular cache, and if there is no hit, the hash is added to the regular cache, but the same hash is then looked up in the feedback cache. If either of the cache lookups results in a hit, the hash is sent to the dd-dst. Otherwise, the original data segment is sent as-is. When a hash encounters a hit in the feedback cache, the hash is deleted after its first use since a corresponding entry would have been made into the regular cache.

4. SOLUTION DETAILS

This section presents the details of the asymmetric caching solution. We first describe its operations at the dd-src and the dd-dst respectively, and then discuss system details for the solution including packet formats and software architecture.

4.1 Operations at the dd-src (SGSN)

The operations of the asymmetric caching at the dd-src can be explained in two parts: downstream and upstream (see Algorithm 1). For every downstream packet, asymmetric caching first divides the packet into value based chunks using Rabin Fingerprinting (line 2). These chunks are then deduplicated using the $regular\_cache$ and $feedback\_cache$ (lines 3 to 10). Specifically, if a matching hash is found in either of the caches, the original chunk is replaced with a shim header and hash of the chunk. Otherwise, it is replaced with a shim header and the original chunk. For every new chunk, its hash is added to the $regular\_cache$ for future use. The new packet is then sent to the dd-dst.

Next, for an upstream packet, if it carries feedback from the dd-dst, asymmetric caching extracts all the hashes and inserts them into the $feedback\_cache$ (line 14 to 16). If the upstream packet was a piggybacked packet, the packet stripped of the feedback is forwarded upstream.

4.2 Operations at the dd-dst (mobile)

The asymmetric caching algorithm at the dd-dst can be explained in terms of three functions: cache maintenance, cache organization, and feedback selection.

First, for cache maintenance, asymmetric caching at the dd-dst (dd-dstDedup) maintains a local cache, called dd-dst_*cache*, that keeps chunks indexed by their hashes and all the flowlets seen thus far. When a packet is received from the dd-src, the dd-dst first reconstructs the original packet (line 3 to 10). The incoming packet is parsed into hashes and clear content by looking at the shim headers. The hashes are replaced by their corresponding chunks found in the dd-dst_*cache*, while clear content is copied as it is, without any shim headers. The reconstructed packet is sent up the network stack. The dd-dst then hashes all the Rabin chunks of the packet to create a list seg_hashes (lines 12 and 13).

Next, for cache organization, asymmetric caching uses the list of hashes, created and maintained by the above mechanism, to select relevant feedback for the ongoing flow. Specifically, the dd-dst first checks if a new flowlet has started in the current flow. This is done by the $updateFlowlets$ module, which uses the segmentation approach presented in Section 3.3. It checks if a statistical boundary has occurred right before the current packet (line 6 and 7). If yes, a new flowlet is added in the current flow and the last flowlet of the flow is linked to it (line 8 to 11). The $gain$ method is used to compute the power of white noise error term of any given series (i.e., sequence of bytes). The flow byte series is modified to remember the content of this packet (line 12). In case this is the first packet, a new flowlet is created for the ongoing flow and the flow byte series is set to be the packet's content (line 1 to 5)

Once the dd-dst has updated the current flowlet in the flow, the dd-dst can select and advertise feedback (*feedback selection*) using seg_hashes. To reduce the complexity of searching in the cache, the dd-dst_*cache* maintains a mapping from each hash to a list of past flowlets in which it had appeared. Only the past flowlets that have seen any of the hashes in the current flowlet are considered for feedback. After extracting these, seg_hashes are inserted in the dd-dst cache along with the current flowlet ID (line 17). Next, the $bestMatchedFlowlet$ module takes the hashes of the packet and determines the number of hits seen in any past flowlet in the cache. The hit count for each past flowlet with the current flowlet is updated using the $subsequenceMatch$ method. This method searches for the seg_hashes among $(win\_factor \times num\_hashes)$ hashes in the past flowlet after the last matched hash in that past flowlet (line 5). The algorithm remembers where the current flowlet has last matched with the old flowlet. After these updates, the $bestMatchedFlowlet$ selects the old flowlet with overall maximum hits for feedback. In case the best matching flowlet is the current flowlet itself, the module selects the second best matching flowlet that has $K\%$ of the maximum hits (line 5 to 9). Feedback is then selected from this old flowlet by the method $selectAdvertisement$.

The $selectAdvertisement$ method keeps track of the last hash that matched between current flowlet and any old flowlet and also the last advertised hash for the pair. For the best matching flowlet, it first determines the later of these two pointers and then jumps δ hashes after that. This δ is the temporal offset to account for the feedback delay incurred by the underlying network. The larger

Algorithm 2 Operations at the `dd-dst`

Input: in_packet = Packet received
Variables:
d_thresh = Distance threshold to determine start of new flowlet
p = Order of AR() model to fit on the series
ar = Estimated coefficients of the p order AR model
win_factor = Region to be searched in the $past\_flowlet$
`dd-dst_cache` = Extensive cache at `dd-dst`
out_packet = Packet to be sent
on_flow = Flow to which in_packet belongs
id_count = Number of flowlets seen so far, used as flowlet id
$current\_flowlet[on\_flow]$ = Latest flowlet being created from on_flow
$hit\_count[flowlet][past\_flowlet]$ = Redundant bytes between current flowlet and an old flowlet in `dd-dst_cache`
$parsed\_pkt$ = Mixed list of chunks and hashes in dedup packet
seg_hashes = List of hashes of chunks
adv_hashes = List of hashes to advertise
$last\_match[flowlet][past\_flowlet]$ = Pointer to last matching hash in $past\_flowlet$ for $flowlet$
$last\_adv[flowlet][past\_flowlet]$ = Pointer to last hash advertised from $past\_flowlet$ for $flowlet$
δ = Temporal offset to account for network delays
Functions:
$hash(chunk)$: Compute hash of $chunk$
$unhash(element)$: Fetch $chunk$ (from cache) whose hash is $element$
$rabinFingerprints(string)$ = Return value based chunks of $string$

gain(bseries)
1: $N \leftarrow len(bseries)$
2: covariance matrix $C \leftarrow [C_{i,j}]$, for $0 \leq i,j \leq p$, where $C_{i,j} \leftarrow \frac{1}{N-p}\sum_{k=p}^{N} bseries[k-i] * bseries[k-j]$
3: matrix $D \leftarrow [D_{i,j}]$, for $0 \leq i,j \leq p-1$, where $D_{i,j} \leftarrow C_{i+1,j+1}$
4: column vector $b \leftarrow [b_{i,0}]$, for $0 \leq i \leq p-1$, where $b_{i,0} \leftarrow C_{i+1,0}$
5: $\alpha \leftarrow D^{-1} * b$ ▷ estimate AR coefficients
6: vector $ar \leftarrow [1, \alpha_0, ..., \alpha_{p-1}]$
7: **return** $N * log(ar * C * ar^T)$

updateFlowlets(on_flow,packet)
1: **if** $series[on\_flow]$ is $null$ **then**
2: $series[on\_flow] \leftarrow packet$ ▷ First packet in the flow
3: $current\_flowlet[on\_flow] \leftarrow (++id\_count)$
4: **return**
5: **end if**
6: $d_{packet} \leftarrow gain(series[on\_flow] + packet) - gain(packet) - gain(series[on\_flow])$
7: **if** $d_{packet} > d_{thresh}$ **then**
8: $last\_flowlet \leftarrow current\_flowlet[on\_flow]$
9: $current\_flowlet[on\_flow] \leftarrow (++id\_count)$
10: link $last\_flowlet$ to $current\_flowlet[on\_flow]$
11: **end if**
12: $series[on\_flow] \leftarrow packet$
13: **return**

subsequenceMatch(seg_hashes, flowlet, past_flowlet)
1: $num\_hashes \leftarrow len(seg\_hashes)$
2: **if** $last\_match[flowlet][past\_flowlet]$ is $null$ **then**
3: $last\_match[flowlet][past\_flowlet] \leftarrow$ first hash in $past\_flowlet$
4: **end if**
5: $max\_seq \leftarrow$ hashes in seg_hashes found among $win\_factor * num\_hashes$ hashes after $last\_match[flowlet][past\_flowlet]$
6: $last\_match[flowlet][past\_flowlet] \leftarrow$ last hash in max_seq
7: **return** length of max_seq

bestMatchedFlowlet(flowlet,seg_hashes,old_flowlet_list)
1: **for** each $past\_flowlet$ in $old\_flowlet\_list$ **do**
2: Add $subsequenceMatch(seg\_hashes, flowlet, past\_flowlet)$ to $hit\_count[flowlet][past\_flowlet]$
3: **end for**
4: $best\_past \leftarrow past\_flowlet$ with maximum $hit\_count[flowlet][past\_flowlet]$
5: **if** $best\_past$ is $flowlet$ **then** ▷ Ongoing flowlet matches most
6: $top2 \leftarrow past\_flowlet$ with 2^{nd} most hits

Algorithm 2 Operations at the `dd-dst`(continued)

7: **if** $hit\_count[flowlet][top2] \geq K\%$ of $hit\_count[flowlet][best\_past]$ **then**
8: $best\_past \leftarrow top2$
9: **end if**
10: **end if**
11: **return** $best\_past$

selectAdvertisement(flowlet,matched_flowlet)
1: $anchor \leftarrow$ later of $last\_match[flowlet][matched\_flowlet]$ and $last\_adv[flowlet][matched\_flowlet]$
2: $anchor \leftarrow \delta + 1$ hash after $anchor$
3: $adv\_hashes \leftarrow MTU/(2 * hash\_length)$ hashes after $anchor$ in $matched\_flowlet$
4: Remove all hashes from adv_hashes which have been seen downstream or sent upstream before **return** adv_hashes

dd-dstDedup(in_packet)
1: **if** in_packet is coming from `dd-src` **then**
2: $on\_flow \leftarrow$ (src ip, dest ip, src port, dest port) of $packet$
3: $parsed\_pkt \leftarrow$ Parse in_packet into chunks and hashes
4: **for** each $element$ in $parsed\_pkt$ **do**
5: **if** $element$ is a hash **then**
6: $out\_packet \leftarrow out\_packet + unhash(element)$
7: **else**
8: $out\_packet \leftarrow out\_packet + element$
9: **end if**
10: **end for**
11: send out_packet to the application above
12: $pkt\_chunks \leftarrow rabinFingerprints(out\_packet)$
13: $seg\_hashes \leftarrow$ list of hashes of pkt_chunks
14: $updateFlowlets(on\_flow, out\_packet)$
15: $flowlet \leftarrow current\_flowlet[on\_flow]$
16: $old\_flowlet\_list \leftarrow$ all the $past\_flowlet$ in `dd-dst_cache` in which any of seg_hashes have appeared
17: Insert seg_hashes with $flowlet$ in `dd-dst_cache`
18: **if** $old\_flowlet\_list$ is $null$ **then**
19: $adv\_hashes \leftarrow null$
20: **else**
21: $best\_flowlet \leftarrow bestMatchedFlowlet(flowlet, seg\_hashes, old\_flowlet\_list)$
22: $adv\_hashes \leftarrow selectAdvertisement(flowlet, best\_flowlet)$
23: **end if**
24: **else** ▷ packet is going to `dd-src`
25: **if** adv_hashes is not $null$ **then**
26: $out\_packet \leftarrow in\_packet + adv\_hashes$
27: **end if**
28: send out_packet to the `dd-src`
29: **end if**

the feedback delay, the more the offset must be for the feedback to be relevant when it reaches the `dd-src`. In our implementation we derive δ from the uplink and downlink data rates seen at the `dd-dst`. After including the temporal offset, the `dd-dst` selects $MTU/(2{*}hash\_length)$ hashes to advertise to the `dd-src`. We choose these many hashes as the minimum chunk size created by our $rabinFingerprints$ method is $2 * hash\_length$, so $MTU/(2 * hash\_length)$ is the maximum number of chunks expected in a downstream packet.

This feedback is further optimized by removing from it all hashes that have occurred in the downstream or have been advertised upstream by the `dd-dst`. Note that if there are no hits in this packet, no feedback is generated (line 18 of `dd-dstDedup`). The feedback to the `dd-src` is opportunistically piggybacked on upstream data packets, e.g. TCP ACKs. If the `dd-dst` has some feedback to send in the form of hashes, it inserts these hashes into the payloads of upstream packets (line 26 in `dd-dstDedup`) and sends the packet upstream. If upstream data packets are not pending to be transmitted, a custom packet is constructed and transmitted.

Table 1: Performance of hash algorithms in collision handling

Hash algorithm	Digest size	Data set size	Collisions	TCP checksum detection
SHA1	20B	5GB	0	N/A
MD5	16B	5GB	0	N/A
Jenkins-8B	8B	5GB	0	N/A
Jenkins-4B	4B	5GB	0.02%	100%

4.3 Related Issues

Although the detailed algorithms presented above have established key insights for asymmetric caching, there are several issues associated with the design choices of the algorithms.

- *Hash function selection*: We use Bob Jenkins hash algorithm to create 8B hashes of the packet chunks on the `dd-src` and `dd-dst` [18]. The choice of hash is based on two conflicting goals: desire for more bandwidth savings by reducing the number of bytes sent on the network and minimum collision rate so that packets are not corrupted during dedup. Popular hash functions such as SHA1 and MD5 are typically computationally heavy and the digest size is large. We compare the aforementioned hash algorithms and the two versions of Jenkins hash (a 4B digest size and an 8B digest size) to hash the Rabin fingerprints generated over $5GB$ of traces. As shown in table 1, we observe no collisions with SHA-1, MD5 and Jenkins 8B hash, but 0.02% collision rate with Jenkins 4B hashing. Jenkins 8B provides a good trade-off between bandwidth savings and collisions, making it our choice for the implementation. Jenkins hash has also been used in prior work on network dedup [3, 14]
- *Handling hash collisions*: We use TCP checksum to detect hash collisions. A TCP checksum is computed on the header and the payload of a packet. As we do not mangle TCP headers, the checksum is sent unchanged. After reconstructing the original packet from a deduped packet the `dd-dst` device checks to see if the checksum matches. If not, it sends a control message upstream to the `dd-src`, requesting it to delete all the hashes that it had sent in the corresponding packet. The `dd-dst` also includes the hashes in the upstream packet. In our hash algorithm evaluation, we also apply this detection test to all cases. While the test was not invoked with SHA1, MD5 or Jenkins 8B, it was invoked by the Jenkins 4B hash. Table 1 shows that TCP checksum was able to detect 100% of the collision events.
- *Cache management*: Both the `dd-src` and `dd-dst` have finite cache space. Thus, asymmetric caching uses an LRU cache eviction policy. At the `dd-src`, LRU runs independently on the regular cache and the feedback cache, and evicts the least recently used hashes in each cache. As the `dd-dst` cache is organized in flowlets, the LRU on the `dd-dst` evicts least recently used flowlets at every run. All the state maintained for that flowlet is removed. The hashes (and chunks) seen in that flowlet are also removed unless they have also appeared in some other flowlet, which is still in the cache. A hash (and the original chunk) is evicted once the last flowlet referencing it is removed from the cache.

4.4 System Architecture

In the rest of this section we present a system architecture for asymmetric caching that is detailed in Figure 2. As shown in the figure, asymmetric caching (AC) works at layer 2.5 on the `dd-src` and `dd-dst`.

- *Dedup source:* This module (shown in the dotted-line box in Figure 2 (a)) captures the downstream packets at the base-station. The captured packet is then broken into chunks using Rabin Fingerprinting and hashes of each chunk are searched in the regular and feedback cache. If a matching hash is found, that chunk is replaced with its hash in the packet. If any packet is modified from its original, the IP options value is changed to reflect the same. We consider transport layer payload for dedup. Inside the transport layer payload, 2B[5] shim headers are inserted before every individual chunk and hash to demarcate original content from hashed content. Shim headers are not added if the entire packet is to be sent as original. This modified packet is then inserted back in the stack to be routed out.

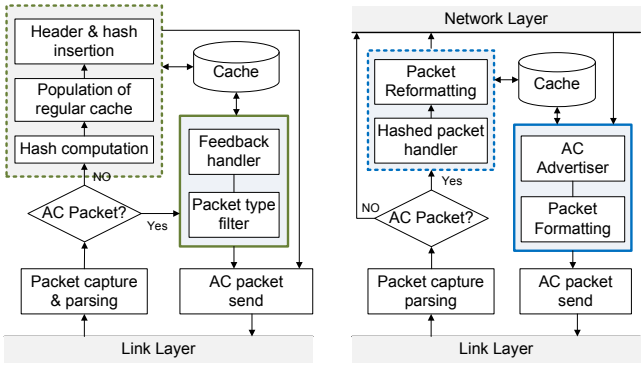


Figure 2: Software prototype of asymmetric caching: It consists of `dd-src` at 2.5 layer of the base station and `dd-dst` at 2.5 layer of the mobile.

- *Dedup receiver:* At the mobile, the dedup packet is received by the receiver module (the dotted-line box in Figure 2 (b)). This module takes care of inflating the packet into its original form, updates flowlets in the caches, and selects the hashes to be advertised. It then passes the reconstructed packet to the higher layer (i.e., network layer).
- *Feedback source:* This module on the mobile device (the thick solid line box in Figure 2 (b)) is responsible for getting the hashes chosen by the dedup receiver and piggybacking the hashes on the next upstream packet. This module captures an outgoing packet, using Netfilter[19], modifies the IP options field and adds feedback to the packet. This packet is then inserted into the stack to be routed out.
- *Feedback receiver:* At the base-station, the feedback receiver (the thick solid line box in Figure 2 (a)) captures upstream IP packets with header options set. It then strips off the feedback from the payload, restores the original header and forwards the new packet to upstream nodes. The extracted hashes are inserted into the feedback cache at the base-station and used for further network deduplication.

5. PERFORMANCE EVALUATION

We evaluate asymmetric caching via trace-based analysis of real network traces. We first explain the trace collection environment, and then describe the trace analysis methodology. Finally, we present trace-based evaluation results as well as prototype-based experimental results.

[5]The first bit is set if it is a hash value and the rest of the bits indicate the length of the following chunk/hash

5.1 Collecting Network Traffic

We use real network traffic collected from 30 different mobile users (volunteers), 5 of whom are smartphone users and the rest are laptop users. We use these traces for performing the trace-based evaluation. Below are the details of the trace collection process.

- *Connectivity:* The laptop users relied only on WiFi connectivity for their network access. The smartphone users relied on both WiFi and 3G connectivity. The data collection spanned a period of 3 months and yielded over 26 $Gigabytes$ of unsecured downlink data. Since we do not require any change in the user access pattern for the trace collection, users accessed the internet as per their normal behavior.
- *Devices and tools:* The laptop users ran Windows 7 and Linux operating systems and used Wireshark to collect their traces. The smartphone users used the Samsung Vibrant Galaxy phone and the HTC G2 phone, both running the Android 2.1 operating system on the T-Mobile network and relied on Tcpdump for trace collection. Users were able to parse trace files and remove any sensitive information before submitting them for analysis. Only unencrypted traffic was used in the analysis.
- *User demographics:* The volunteers included full-time employees at an industry research lab and an enterprise, as well as graduate students at a large university campus. The users span the age group of 21 to 50 and were spread over two different geographic regions.

5.2 Analysis Methodology

We use a custom trace analyzer to operate on the above traces. The analyzer models components of asymmetric caching presented in Section 4 and is configured and used for analysis, as follows:

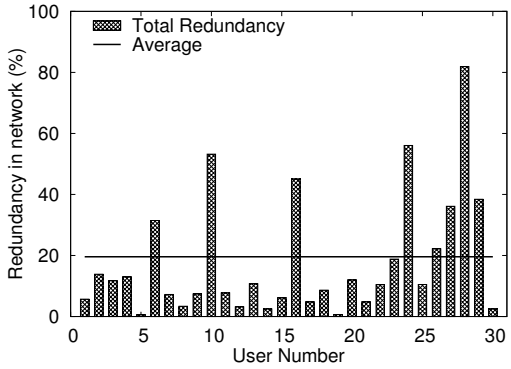
Caches: The analyzer maintains three caches in the form of hash tables (i) `dd-src` regular cache (ii) `dd-src` feedback cache and (iii) `dd-dst` cache. It also maintains additional data structures required by the asymmetric caching algorithm at the `dd-dst`. We set the default `dd-src` cache size (i.e., regular + feedback) to 1MB and the default `dd-dst` cache size to 250MB. We explicitly study the sensitivity of the solution to cache sizes later in the section.

Past and Present Trace: To emulate the temporal history of the traces, the packet trace for each user is split equally into a past trace and a present trace (e.g., a 40MB trace was split into a 20MB *past* and a 20MB *present*. Then, the past trace is used as an input to the analyzer to populate the initial cache at the `dd-dst`. This is the memory collected at the `dd-dst` without the knowledge of the current `dd-src`. Next, in the present trace, a set of 30 connections is randomly selected and used for the second set of inputs to the analyzer.[6] We use a minimum threshold of 5KB for the size of connections under consideration to avoid very small (redundant) connections and to filter out insignificant connections from the analysis.

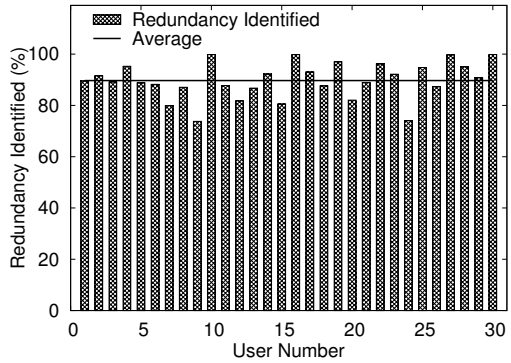
Metrics: Given the above set-up, we monitor three values for each user: (1) redundant bytes found in the `dd-src`'s regular cache, (2) redundant bytes found in the `dd-src`'s feedback cache and (3) total number of unique hashes sent as feedback from `dd-dst` to the `dd-src`.

Comparison: We also implement and run the byte-sequence caching algorithm (with average segment size of 128B) on a merged trace of the *p*ast and the 30 connections from the present for each user. This represents the scenario where the `dd-src` has all the hashes that the `dd-dst` has ever seen and thus gives a measure of 'ideally' achievable dedup.

[6]We have observed that the trend of redundancy is similar even when considering the entire present trace for each user.



(a) Total network redundancy



(b) Redundancy identified

Figure 3: Identifying network redundancy: (a) shows there exist network redundancy (avg., 19.6%) and (b) shows asymmetric caching finds most of network redundancy (avg., 89.7%).

5.3 Evaluation Results

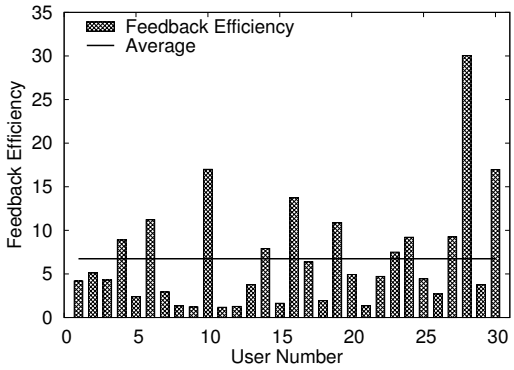
5.3.1 Identifying Network Redundancy

We first show how much network redundancy exists in the collected traces and how much of that redundancy can be identified by asymmetric caching. Figure 3(a) plots total network redundancy that exists in the 30 user traces. Here, the total network redundancy is defined as the number of total cache hits at `dd-dst` for each chunk of a received packet, if the chunk exists in local cache (cache hit), we count the chunk as redundant bytes. As shown in the figure, there is indeed network redundancy of 19.6% on average.

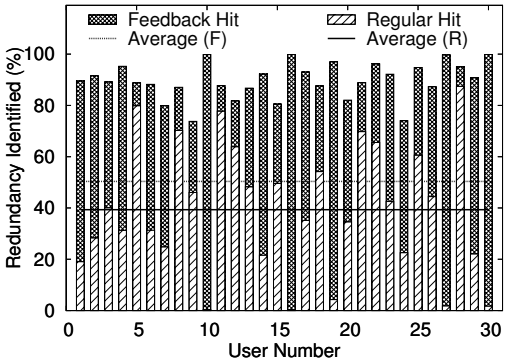
Next, Figure 3(b) shows the percentage of the redundant bytes identified by asymmetric caching (found in Figure 3(a)). Specifically, we measure the number of cache hits at a `dd-src` based on both regular and feedback caches and use it for the redundant bytes identified. As shown in the figure, the asymmetric caching is able to identify on average 89.7% of the total redundancy, a considerable fraction of which is attributable to the feedback (see next subsection). Also note that its variance is small across 30 different users, owing to our fine-grained and adaptive advertisement scheme.

5.3.2 Feedback Efficiency

In this section, we present the feedback efficiency of asymmetric caching. Figure 4(a) plots the ratio (λ) of the redundant bytes

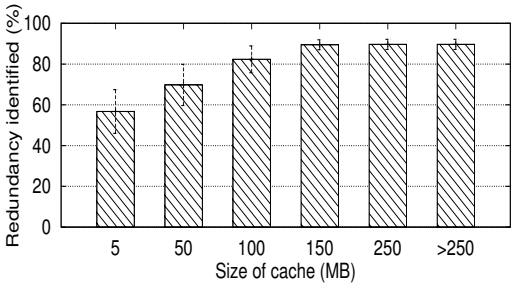


(a) Redundancy to feedback

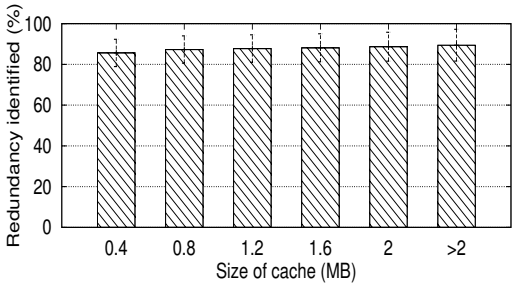


(b) Feedback hits vs Regular hits

Figure 4: Feedback Efficiency: (a) shows the ratio of the total redundancy identified to the feedback bytes (λ), and (b) shows how much each cache (feedback and regular) contributes to the redundancy detection.



(a) Sensitivity to cache size at `dd-dst`



(b) Sensitivity to cache size at `dd-src`

Figure 5: Sensitivity to cache size: (a) shows that asymmetric caching can identify 89% of redundancy by using 150MB on the mobile device. (b) shows that the asymmetric caching requires only a small cache size at the `dd-src` (e.g., ∼1MB) to achieve 85% of the redundancy detection.

identified to every feedback byte. If λ is greater than 1, asymmetric caching's feedback is effective in finding redundancy, and vice versa. As shown in the figure, the average λ value over 30 users is 6.74. One interesting observation is that the higher a user shows mobility (i.e., smartphone users including user 10, 28, 30), the higher λ is.

Next, we further study how much of the redundancy elimination is attributable to the feedback cache versus the regular cache. Recall that the redundant bytes are identified by searching its regular cache and then, if not found there, the feedback cache. For this, we analyze the redundancy found using only the feedback cache (F) and the redundancy found using only the regular cache (R). Figure 4(b) shows relative contribution from both R and F for each user. As shown in the figure, feedback cache largely contributes to 50.35% of the redundancy elimination, whereas the regular cache contributes to 39.3% of the redundancy elimination. Note that the hits in the regular cache can be considered as an indication of the performance of conventional dedup that relies only on symmetric caching. Hence, this result shows that asymmetric caching can improve the performance of dedup significantly.

5.3.3 Sensitivity to Cache (Memory) Size

In this section we measure the sensitivity of asymmetric caching to cache size. In this experiment, we first fix the cache size of the `dd-src` to 2MB. Then, while we increase the cache size of the `dd-dst` (from 5MB to >250MB), we analyze the percentage of redundancy identified by the asymmetric caching. We also analyze the opposite setting to study the sensitivity of `dd-src`'s cache size on its performance (250MB of `dd-dst` cache and from 0.4MB to >2MB of `dd-src` cache).

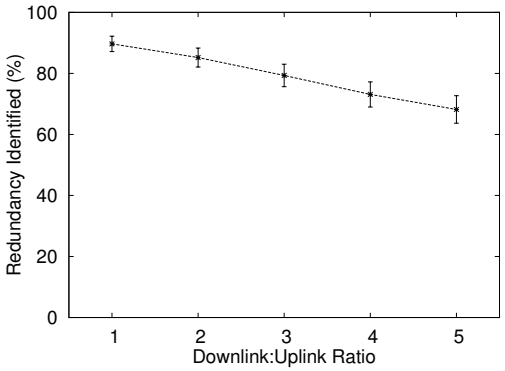
Figure 5 shows the results of the both experiments. First, as shown in Figure 5(a), given the constant size of cache at the `dd-src`, the redundancy identified by asymmetric caching increases with the increase in `dd-dst` cache size. This trend is a result of fewer cache evictions when using larger caches. In addition, with only 150MB of cache size at `dd-dst`, asymmetric caching is able to identify 89% of the redundancy.

Next, as shown in Figure 5(b), even with a small cache (e.g., ∼1MB) at `dd-src`, asymmetric caching is able to identify more than 85% of redundancy. Finally, looking at the both figures in Figure 5, we can observe that for a 1:100 ratio in the cache sizes (e.g., ∼1MB at the `dd-src` and 100MB at the `dd-dst`), asymmetric caching effectively detects and leverages redundancy. Furthermore, this ratio supports the design goal of asymmetric caching—the use of large cache at the `dd-dst` with a small cache requirement at the `dd-src`.

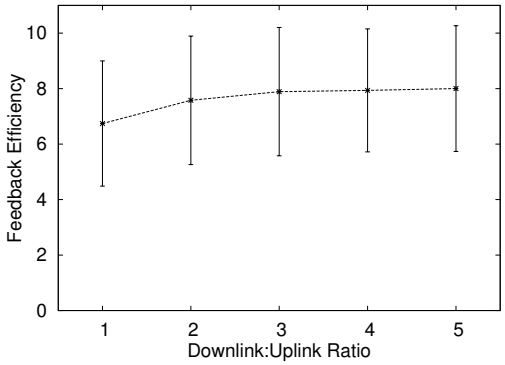
5.3.4 Performance under varying data-rates

So far we have assumed that the uplink and downlink data-rates are same, i.e. for every byte downstream, the mobile sends a byte of feedback upstream. In this section, we analyze the performance of asymmetric caching when the downlink and uplink data-rates are asymmetric. We vary the ratio of downlink to uplink data-rates from 1 through 5 and monitor the percentage of redundancy leveraged and feedback efficiency in each case. The results are shown in Figure 6.

We observe in Figure 6(a) that as downlink rate grows to 5× the uplink rate, the redundancy identified by asymmetric caching



(a) Redundancy leveraged



(b) Feedback efficiency

Figure 6: Performance with varying data-rates: (a) shows the % redundancy identified for increasing downlink to uplink data-rate ratio, and (b) shows the feedback efficiency in each case.

goes down to 68%. Interestingly, for the same ratio, the feedback efficiency grows to 8×, as shown in Figure 6(b). This shows that when asymmetric caching is restricted to send less upstream feedback, the redundancy elimination performance drops modestly and the feedback is more efficient.

5.3.5 Application-agnostic dedup

Finally, we measure how effectively asymmetric caching works for different types of application traffic. Recall that one of the design principles of asymmetric caching is to remain application agnostic— not require application information— but provide dedup performance regardless of application types. To this end, we analyze the trace of 30 users to calculate the percentage of redundancy identified by asymmetric caching over total network redundancy for each application. We classify the applications based on the port numbers used for the connections.

Table 2 shows the performance results of asymmetric caching

Table 2: Application-agnostic redundancy identification

Applications	Port Numbers	Redundancy Identified (%)
HTTP	80	98.76
ICSLAP	2869	80.48
Android Market	5228	53.10
McAfee, HP, SAP	5555	82.13
P2P and others	Ephemeral	97.70

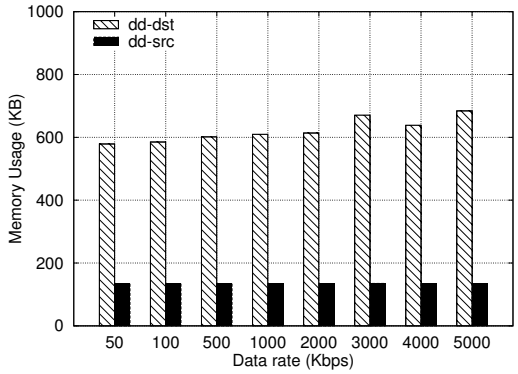


Figure 7: Memory consumed by Asymmetric Caching

under different types of applications. As shown in the table, the asymmetric caching is able to identify 53% to 97.76% of the network redundancy and it does not have radical performance penalty for specific types of application. This clearly supports our design goal of application-agnostic feature.

5.4 Prototype Results

To further validate the feasibility of asymmetric caching, we implement asymmetric caching in our testbed and have evaluated its CPU and memory overhead. We implement the `dd-src` on a desktop running Ubuntu OS with a 2GHz dual-core processor and 2GB memory. The desktop is equipped with an Atheros chipset based network card (NIC) with Madwifi [20] driver and talks to the mobile over wireless link. Using Madwifi, the `dd-src` is set in master mode. The desktop is connected to another desktop, which serves as server, over a wired link.

Next, the `dd-dst` is implemented on a Samsung Nexus S smartphone with 512MB RAM and 1GHz pocessor, running Android OS. The `dd-dst` has been implemented as a user space Linux module on the Android system. We use Netfilter framework on Linux to capture and modify downstream and upstream packets.

We use Iperf [21] to create packets with random content and send them over the network. We run Iperf at different data rates (from 50Kbps to 5Mbps) for 60s, and record the CPU and memory footprint of asymmetric caching on both `dd-src` and `dd-dst`, using Top utility. Our experiments show that the memory footprint of asymmetric caching is small enough to be deployed on a mobile phone and base-station. As shown in Figure 7, the physical memory taken by the `dd-dst` process stays around 600KB in our experiments while at the `dd-src` it is even lower at 140KB. This supports our goal of 'asymmetric' cache sizes, reducing the overhead on a `dd-src` serving millions of mobile devices.

In terms of CPU consumption, Figure 8 shows that the CPU usage on the mobile phone grows with growth in data-rate, but stays 15% even at 5Mbps data-rate. The CPU usage on the `dd-src` stays very minimal, less than 1% throughout. The CPU usage on the mobile increases with the data rate as much larger caches have to be searched to generate relevant feedback.

Overall, our user space implementation shows that asymmetric caching is feasible on both the ends of dedup identified in our motivational scenarios.

6. RELATED WORK

Network dedup approaches: The notion of network dedup was first presented in [7], where packets are decomposed into segments

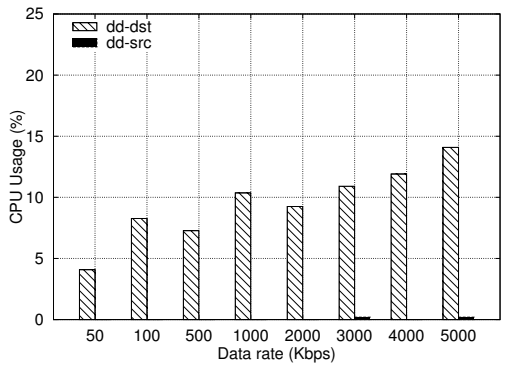


Figure 8: CPU occupied by Asymmetric Caching

using the Rabin fingerprinting algorithm so that partial-packet redundancy can be exploited. This approach was developed further in value-based web caching [1], where the data is cached on its value rather than its name. The idea of using packet caches on routers was introduced in [13]. In [2], the authors perform an experimental study of redundancy across 12 different enterprise networks. Both [13] and [2] identify that significant bandwidth savings can be achieved by using packet level dedup approaches, thereby motivating the current work. Similarly, EndRE [3] is an end-to-end solution for network dedup, which presents a new fingerprinting scheme called SampleByte. Recently, [14] proposed overhearing content in wireless networks to dedup across wireless users. Celleration [22] is another sender-driven dedup solution that leverages inter-user redundancy for a single point of attachment. All the above works are designed for static scenarios, do not work across points of attachment and do not support IP address changes due to mobility. Asymmetric caching is complementary to the above works and specifically optimizes wireless traffic, without requiring modification to internet servers.

Application layer dedup: Application layer works include caching http objects on browsers [23] and on proxy servers [24], delta encoding, file differencing (e.g. VCDIFF) [25], techniques for detecting duplicate transfers of the same file [4] and techniques such as base-instant caching [5], template caching [6] for enhanced cacheability of dynamic objects. More recent developments include content-delivery networks [26] and peer-to-peer caching solutions [27]. All these solutions operate at the granularity of files or application-objects and hence do not provide fine-grained redundancy elimination. Further, they are application layer solutions and have to be realized independently for every single application. Most importantly, using proxy or other intermediate caches while reducing the load on servers does not reduce the traffic on the wireless link. Asymmetric caching operates agnostic to different applications.

Transport layer dedup: Recently, Zohar *e*t al [28] propose an end-to-end receiver driven dedup solution that extends TCP options. The receiver matches TCP stream chunks with its cache and sends predictions for future chunks in the ongoing flow to the sender. The dedup solution in [28] is similar to asymmetric caching in that both use receiver driven feedback to improve dedup performance. However, there are fundamental differences. The solution in [28] is closely tied to the TCP protocol and operates at a coarse data-granularity. Asymmetric caching on the other hand is transport protocol agnostic and operates at sub-packet level granularity. The solution in [28] is an end-to-end solution, whereas asymmetric caching is a last hop solution. This is important as wireless service providers who have the motivation to utilize their spectrum better can deploy asymmetric caching without any dependencies on the content provider. Also, the solution in [28] does not partition old connections into flowlets and hence maintains connections in their entirety. However, asymmetric caching partitions content into flowlets depending on their stationarity and this helps when the composition of connections changes in terms of a few objects or in terms of the ordering of the objects. Finally, the solution in [28] does not address how feedback might be chosen when chunks experience hits with multiple old connections. The feedback selection algorithm in asymmetric caching explicitly tackles this problem by choosing from multiple flowlets. This capability is especially important when operating at fine data granularities.

7. CONCLUSION

In this paper, we propose *asymmetric caching*, an improvement to baseline network deduplication that allows the $dedup\ destination$ to selectively feedback appropriate portions of its cache to the $dedup\ source$ with the intent of improving the redundancy elimination efficiency. We show using traffic traces collected from 30 mobile users, that with asymmetric caching, over 89% of the achievable redundancy can be identified and eliminated *even when the* $dedup\ source$ has less than *one hundredth of the cache size as the* $dedup\ destination$. Further, we show that the number of bytes saved from transmission at the $dedup\ source$ because of asymmetric caching is over $6\times$ that of the number of bytes sent as feedback. Finally, with a prototype implementation of asymmetric caching on both a Linux laptop and an Android smartphone, we demonstrate that the solution is deployable with reasonable CPU and memory overheads.

8. ACKNOWLEDGEMENTS

The authors would like to thank Mr. Nitin Agarwal at the University of Illinois for early discussions and pointers on techniques to identify changes in stationarity in a time-series. The authors would also like to thank Dr. Ulas Kozat for acting as a shepherd for the camera-ready version of the paper.

9. REFERENCES

[1] Sean C. Rhea and Kevin Liang. Value-based web caching. In *The 12th Int. World Wide Web Conference*, 2003.

[2] Ashok Anand, Chitra Muthukrishnan, Aditya Akella, and Ramachandran Ramjee. Redundancy in network traffic: findings and implications. In *ACM SIGMETRICS*, 2009.

[3] Bhavish Aggarwal, Aditya Akella, Ashok Anand, Athula Balachandran, Pushkar Chitnis, Chitra Muthukrishnan, Ramachandran Ramjee, and George Varghese. Endre: an end-system redundancy elimination service for enterprises. In *NSDI*, 2010.

[4] Yee Man Chan, Terence Kelly, and Jeffrey C. Mogul. Design, implementation, and evaluation of duplicate transfer detection in http. In *NSDI*, 2004.

[5] Barron C. Housel and David B. Lindquist. Webexpress: a system for optimizing web browsing in a wireless environment. In *ACM MobiCom*, 1996.

[6] Fred Douglis, Michael Rabinovich, and Antonio Haro. Hpp: Html macro-preprocessing to support dynamic document caching. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[7] Neil T. Spring and David Wetherall. A protocol-independent technique for eliminating redundant network traffic. In *ACM SIGCOMM*, 2000.

[8] Riverbed, www.riverbed.com/us/solutions/wan_optimization/.

[9] Comscore Report: Digital Omnivore, October 2011. URL: www.comscore.com/Press_Events/Presentations_Whitepapers/2011/Digital_Omnivores.

[10] WeFi Analytics Report, August 2010. URL: mobilemarketingmagazine.com/sites/default/files/WeFi\%20Wi-fi\%20Data\%20Report\%20Q1\%202010.pdf.

[11] Ip core pooling tutorial, august 2006. URL: archive.ericsson.net/service/internet/picov/get?DocNo=1/28701-FGB101256.

[12] Ericsson SGSN-MME. URL: www.ericsson.com/ourportfolio/products/sgsn-mme?nav=fgb_101_256.

[13] Ashok Anand, Archit Gupta, Aditya Akella, Srinivasan Seshan, and Scott Shenker. Packet caches on routers: the implications of universal redundant traffic elimination. In *ACM SIGCOMM*, 2008.

[14] Ashok Anand Shan-Hsiang Shen, Aaron Gember and Aditya Akella. Refactor-ing content overhearing to improve wireless performance. In *ACM MobiCom*, 2011.

[15] EMC Data Domain, www.emc.com/backup-and-recovery/data-domain/data-domain.htm.

[16] M. O. Rabin. Fingerprinting by random polynomials. In *Technical Report TR 15-81, Department of Computer Science, Harvard University*, 1981.

[17] Ulrich Appel and Achim V. Brandt. Adaptive sequential segmentation of piecewise stationary time series. *Information Sciences*.

[18] Bob jenkins hash functions. URL: burtleburtle.net/bob/.

[19] Netfilter, www.netfilter.org/projects/libnetfilter_queue/.

[20] MadWifi Driver, www.madwifi.org.

[21] Iperf, iperf.sourceforge.net/.

[22] Eyal Zohar, Israel Cidon, and Osnat (Ossi) Mokryn. Celleration: loss-resilient traffic redundancy elimination for cellular data. In *ACM HotMobile*, 2012.

[23] RFC 2616: Hypertext Transfer Protocol – HTTP/1.1, www.w3.org/Protocols/rfc2616/rfc2616.html.

[24] The apache HTTP Proxy, httpd.apache.org.

[25] RFC 3284: The VCDIFF Generic Differencing and Compression Data Format, www.faqs.org/rfcs/rfc3284.html.

[26] The Akamai Content Delivery Network, www.akamai.com.

[27] On Saleh and Ma Hefeeda. Modeling and caching of peer-to-peer traffic. In *IEEE ICNP*, 2006.

[28] Eyal Zohar, Israel Cidon, and Osnat (Ossi) Mokryn. The power of prediction: cloud bandwidth and cost reduction. In *Proc. of the ACM SIGCOMM*, 2011.