Squeezing more bits out of HTTP caches

Jeffrey C. Mogul
Compaq Computer Corporation Western Research Laboratory
250 University Avenue
Palo Alto, CA 94301
mogul@pa.dec.com

Computer systems designers often uses caches to solve performance problems. Caching in the World Wide Web has been both the subject of extensive research and the basis of a large and growing industry. Traditional Web caches store HTTP responses, in anticipation of a subsequent reference to the URL of a cached response. Experience with real Web users, unfortunately, shows that there are limits to the performance of this simple caching model: many responses are useful only once. Researchers have proposed a variety of more complex ways in which HTTP caches can exploit locality in real reference streams. This paper surveys several techniques, and reports the results of trace-based studies of a proposal based on automatic recognition of duplicated content.

6 October 1999

Squeezing more bits out of HTTP caches

Computer systems designers often uses caches to solve performance problems. Caching in the World Wide Web has been both the subject of extensive research and the basis of a large and growing industry. Traditional Web caches store HTTP responses, in anticipation of a subsequent reference to the URL of a cached response. Experience with real Web users, unfortunately, shows that there are limits to the performance of this simple caching model: many responses are useful only once. Researchers have proposed a variety of more complex ways in which HTTP caches can exploit locality in real reference streams. This paper surveys several techniques, and reports the results of trace-based studies of a proposal based on automatic recognition of duplicated content.

1. Introduction

HTTP accounts for most of the bytes flowing over the Internet backbone (up to 75%, in one study [52]). This bandwidth demand requires continued investment in link and switch capacity, and leads to congestion, which increases user-perceived latency. At the edges of the Internet, which are often bandwidth-constrained, every extra byte transferred adds incremental delay; this is a particular problem for home users, who do not yet have a cost-effective means to increase bandwidth above 56Kbits/sec. And every round-trip through the Internet adds delay, often several hundred milliseconds.

Almost any computer system that suffers from latency or bandwidth problems can benefit from caching. The Web is no exception, and caching mechanisms have been part of HTTP almost since its inception. Caching is perhaps the one aspect of the Web most easily amenable to academic studies, and many research papers have been published. Web caching is also sufficiently useful to have led to the creation of a moderately large and rapidly growing industry.

Caching works when a reference stream has locality. Temporal locality exists when an item is referenced more than once -- a cache can store the item on the first reference, and then return it for subsequent references. Traditionally, Web caches have exploited temporal locality, with a URL as the granularity of reference. That is, such a cache would store a response to a request for a URL, and then a subsequent request for the same URL yields a cache hit.

Other caches in computer systems, such as CPU data caches, often reach hit rates approaching 100%, but numerous studies of actual Web reference streams report much lower hit rates, often around 50% or less. Indeed, recent studies have shown intrinsic limits to the hits rates achievable with URL-granularity temporal locality: no matter how large the cache or the user population, many references will never be cache hits.

This paper surveys some of the techniques proposed to extend the effectiveness of HTTP caches, by exploiting other forms of locality, or by ensuring coherency requirements without having to defeat caching. These techniques include cooperative caching, prefetching, support for partial transfers, differential cache updates (also known as delta encoding), HTML macros, and several mechanisms for improving or ensuring coherency. The paper also surveys several methods that have been used to evaluate these techniques, and present recent results suggesting that a proposal for automatic duplicate suppression can improve cache utility.

2. Reasons for HTTP caching

Web caches have proved useful for three purposes:

1. **Latency reduction:** a cache can often deliver content to a client faster than the *origin server* (HTTP's term for the original source of the content).

2. **Bandwidth conservation:** whenever a cache avoids the transmission of bytes to or from the origin server, this reduces bandwidth requirements on that portion of the network, resulting in cost savings and lower congestion. Reduced congestion can also improve overall latency.

3. **Disconnected operation:** a cache can provide access to information when the origin server is unavailable, either due to network disconnection or to server failure.

Subsequent sections of this paper, describing various techniques for improving HTTP caching, will indicate how each approach affects the first two of these metrics.

Relatively few Web caches truly support disconnected operation, because this would require anticipating all (or almost all) of a user's future requests. While several projects have demonstrated impressive results using caches to support disconnected or weakly-connected distributed file system access [28, 41], these results may prove hard to transfer to the HTTP protocol. (Web caches designed for latency and bandwidth reduction can still, as a side effect, reduce the likelihood that a cachable resource is perceived as inaccessible.)

2.1. Cost and benefit synergies and tradeoffs

Sometimes the motivations for caching conflict: the ultimate user might prefer reduced latency, while a corporate accountant might prefer reduced bandwidth charges. A cache can reduce latency by avoiding network round trips, by avoiding the use of low-bandwidth links, or by prefetching content. Prefetching imposes a tradeoff: a successful prefetch pleases everybody (reducing latency for the user without moving unnecessary bytes), but most prefetching mechanisms often fetch unnecessary bytes as well, increasing the overall bandwidth requirements.

All caching mechanisms also impose a tradeoff between the idealized benefits of caching, and the practical costs of building and operating a computer system. These costs include CPU cycles, RAM, disk storage, and other hardware costs; software costs, especially as implementation complexity increases; and operational costs, particularly when cache algorithms require expert intervention to maintain effectiveness. These costs are not limited to the caches themselves; many HTTP mechanisms intended to support caching impose additional costs on origin servers, and some impose added costs on clients.

Server and cache operators often resist spending more on hardware and software. However, hardware capacities are still increasing at exponential rates (for CPU speeds, RAM sizes, and disk capacity), while the speed of light (and hence round trip times) remains a constant. Bandwidth, unlike latency, is not intrinsically limited, although it is not universally improving (especially not on wireless links). Almost any caching mechanism that promises to eliminate round trips will therefore become cost-effective at some time in the future, and caching mechanisms that improve bandwidth without harming latency should also pay off at some point. On the other hand, a caching system that adds much complexity or network communication in order to avoid CPU or storage costs is not likely to be useful in the long term.

We can encapsulate these considerations into four principles to guide the evaluation of HTTP caching proposals:

1. Exploit Moore's law to avoid having to change the speed of light.

2. Avoid roundtrips, unless the payoff is significant.

3. Use extra header bytes, sparingly, to gain efficiency.

4. Use hints to improve the payoff of speculative approaches.

Section 7 will summarize how the various proposals described in this paper address these tradeoffs.

In addition, any realistic proposal must take into account the difficulty of deploying protocol changes in an installed base that already includes tens of millions of users. A protocol that can be incrementally deployed may have significant practical advantages over a theoretically superior protocol that, however, does not prove useful until significant numbers of systems are upgraded.

2.2. Alternatives to caching

Caching is not the only way to avoid sending bytes over a network; content simplification and data compression can also pay off.

Content simplification works, to a point. For example, Web designers can use common sense to reduce page complexity, or special software tools to optimize image coding (e.g., [17]), but the desire for richer user experiences usually prevails over such pragmatism. New content formats, such as Cascading Style Sheets [31], can increase the coding efficiency without reducing expressiveness. But some content, such as medical images, broadcast-quality video, and executable software, cannot be simplified without loss of meaning. And other emerging formats, such as MPEG-1 Layer 3 (MP3) audio [27], provide new bandwidth challenges.

Data compression directly targets the transmission of redundant bits within a single transfer. Existing general-purpose compression algorithms provide significant size reductions, typically reducing text file sizes by a factor of three or more. However, the bulk of Web-related bytes transferred come from content-types, such as images, video, and audio, which are already heavily compressed using highly efficient content-specific compression algorithms, and so the net benefit of applying general-purpose compression to all Web traffic would be relatively small [39]. The trend towards increased use of non-text media further reduces the potential for general-purpose compression in HTTP.

With the increased transmission of new data types (such as Java byte codes and other software), one can expect to see progress on new type-specific compression techniques. For example, general-purpose algorithms can be tuned for improved compression of binaries [57], and special-purpose compression algorithms based on parse trees can do even better [18]. Even so, compression has its limits, and data types such as program binaries inevitably gain complexity with time.

2.3. Other proxy functions

HTTP supports the use of caches directly integrated with the end-client (e.g., a browser), closely associated with a particular server (‘‘server accelerator’’), or integrated with a proxy server. Many proxy systems serve other functions, such as security protection (i.e., as a firewall), censorship, and transcoding [21] (a non-end-to-end form of content simplification). These functions are not intrinsically related to caching, and are outside the scope of this paper.

Proxy systems set up as server accelerators operate on very different reference streams than other HTTP caches; therefore, server accelerators are also outside the scope of this paper.

3. Methods for evaluating HTTP caching mechanisms

Because caching exploits locality in a reference stream, we can evaluate the potential performance of various caching designs by obtaining a reference stream, and then either analyzing the stream or simulating the performance of a caching design on this stream.

The input reference stream can either be a real stream, captured by tracing the references of actual users, or a stream generated by a model of user activity. Several workload generators have been devised either for use in benchmarking proxy caches [1] or for more general uses [4].

Trace-based studies require more effort, to capture and store the trace logs, but they avoid the simplifying assumptions made when parameterizing a workload generator (and, in any case, the parameters for any realistic workload generator must be based on some set of traces.) While a few studies have been able to capture the reference stream generated by instrumented [3] or cacheless [58] browsers, almost all interesting traces have been captured either by logging at proxies, or by capturing network packets and then reassembling the TCP streams.

Inaccuracies are possible even when using traces to drive simulations or performance measurements, because this still requires certain simplifying assumptions. In particular, a cache may alter the response time seen by the users, who in turn might alter their behavior. One can test the modified system under a live load, but because of unpredictable variations in usage patterns, this might mean giving up the repeatability provided by traces or load generators. Consequently, it may be hard to discern whether a change in the system under test leads to a significant performance improvement.

One can bypass the repeatability problem when comparing two or more implementations by running them simultaneously, and randomly splitting the incoming reference stream among the systems under test. In principle, this should eliminate most of the variation in load, so that any difference in performance can be attributed to the implementation differences. In practice, it might be difficult to achieve truly random load splitting, and it might also be difficult to split the load without affecting locality properties.

4. Limits of simple approaches to HTTP caching

Most studies of Web reference streams and proxy caches report the hit ratio (HR) per resource and the *byte hit ratio* (BHR), weighted by the size of the response body (this is also sometimes called the *weighted hit ratio* or WHR). The bandwidth reduction ratio should be similar to the BHR, although the BHR does not account for HTTP and other protocol header overheads. Simulation studies have reported HRs ranging from 30% to 49% and BHRs ranging from 14% to 36% (assuming an infinite cache size) [16, 19]; the variations may be due to differences in user community, geography, or when the traces were obtained. Reports generated from in-service measurements of the NLANR caches [43] show actual BHRs vary tremendously over short timescales.

It seems impossible to significantly increase hit ratios and byte hit ratios above a (fuzzy) threshold, because of several intrinsic aspects of Web reference streams:

Uncachable resources: Some responses cannot be cached; for example, stock quotes, query results, or electronic commerce ‘‘shopping baskets.’’ Other responses could be provided from a cache, save for the serving site’s desire to gather demographic information or advertising revenue.

Particular causes of cache misses include:

- **Coherency misses**: Any system where data items are frequently updated, where caching is employed, and where use of an out-of-data value could lead to erroneous behavior, requires a mechanism to maintain cache *coherency*. Various coherency mechanisms have been developed for distributed systems, often using techniques such as callbacks or leases. HTTP, however, has no way to guarantee coherency for a resource except by disabling caching for that resource.
- **Nonce URLs**: Some sites implicitly defeat caching by generating a unique URL for each reference to a resource. For example, the same advertising banner may appear on millions of pages, but each time the banner appears with a different URL.

Zipf’s law: The Web is so large that many pages will never be referenced more than once in the reference stream seen by any one cache. Most cache hits come from a small set of resources, but many references are made to resources outside this set. Studies have shown that page re-reference frequencies follow a distribution similar to Zipf’s law, in which the relative probability for a reference to the kth most popular page is proportional to $1/k^a$, for some constant a [7, 44, 53]. This implies that in a large universe of pages, most of these pages are extremely unlikely to be referenced twice in the same reference stream. The first reference via a given caching proxy to any given resource is a *compulsory miss*, since the value cannot possibly in the cache.

High rate of change: Many potentially-cachable resources change fairly rapidly [14], which would lead to cache incoherence if responses for these resources were allowed to be cached. If the semantics of the resource does not require absolute coherency, HTTP allows the origin server to limit the cachable lifetime of a response, in the hope that it will expire before the resource is updated (effectively, a form of lease). But this trades improved latency against potential incoherency.

Voelker *et al.* point out that most Web resources appear to change more rapidly than they are re-referenced, unless the reference stream comes from an enormous population [53].

Resource size distribution: Why is the byte hit ratio almost always lower than the simple hit ratio? Williams et al. [55] observed that most references in their traces were for small resources. Breslau et al. [7], working from several trace sets, showed that there is no strong correlation between resource size and access frequency, although the mean size of popular resources is smaller than the mean for unpopular ones. One possible explanation for these observations is that a small set of small resources accounts for most of the cache hits.

4.1. Practical limits on HTTP caches

It is tempting to evaluate an HTTP caching design by simulating an idealized implementation of the design. However, this runs afoul of several practical constraints on HTTP caches.

The most obvious one is cache capacity. Any real cache has finite storage, and once it fills up, it must evict old entries in order to store new ones, following some *replacement policy*. One simple policy is Least Recently Used (LRU), but various studies (e.g., [9]) have proposed other policies that reduce the number of *capacity misses*, relative to LRU, in some cases. A replacement policy also imposes implementation costs (including meta-data storage, update, and lookup costs), and some policies might not be feasible to implement.

In current technologies, the working set of a large HTTP cache does not fit into an economically reasonable amount of RAM; some or all of the cached data must be stored on disk. Disk access adds latency: not only the disk's average access time, on the order of 10 msec, but also queueing delays if the disk is the bottleneck. These queueing delays can be quite large.

If we are using a cache to reduce latency, then we cannot simply evaluate it on the basis of hit ratio. Instead, we must estimate its weighted access latency, using:

$$weighted\_access\_latency = HR \times hit\_cost + (1 - HR) \times miss\_cost$$
$$miss\_cost = retrieval\_latency + cache\_check\_latency$$

where *HR* is the mean hit ratio, *hit_cost* is the mean total cost of processing a cache hit, and *miss_cost* is the mean latency for a cache miss. The *miss_cost* is the sum of two terms: not just the *retrieval_latency* for retrieving the response from the origin server, but also the *cache_check_latency* to determine if the cache entry is present. (The *hit_cost* may also include *cache_check_latency*.)

If the hit ratio is quite high, and the *cache_check_latency* is not too excessive, then the *weighted_access_latency* will be better than the *retrieval_latency* (the latency without caching), and the cache will pay off. However, observed hit ratios are usually well below 50%, and queueing delays at the disks of an overloaded proxy can be quite large. Therefore, the *hit_cost* may be significant, and one cannot simply assume that an HTTP proxy cache will actually improve overall latency. In particular, the disk subsystem must be carefully designed to avoid imposing excessive latencies [33, 35].

The *cache_check_latency* depends mostly on whether the data structures used for the cache lookup can be held mostly in RAM, or whether even the lookup requires a disk access. Also, if the data structure is stored on disk, then lookups must compete with frequent updates for disk bandwidth. Unless the replacement policy has perfect future knowledge, responses that will never be used for cache hits must be stored, and hence the data structure might be updated far more frequently than it is used for lookups.

With modern large-capacity disks, it might not be unreasonable to configure a cache with sufficient disk storage that it achieves almost the hit ratio of an infinite cache. But it may not pay to increase the cache size beyond the point where the lookup data structure no longer fits in RAM, since capacity misses decline only slowly past a certain cache size.

At the moment, storage capacities of RAM and disk are still growing exponentially. These growth rates should work to improve the effectiveness of HTTP caches, but will hardware improve fast enough, or will Web reference rates grow even faster? One study (of a very small and atypical user population) reported that ‘‘potential for caching requested files in the network has declined’’ between 1995 and 1998 [3], but more extensive studies are warranted.

Given that RAM size is increasing exponentially, but disk access times increase very slowly, and most cache hits come from a relatively small subset of the cachable content, at some point it might become feasible to build RAM-only HTTP proxy caches. That is, if the replacement policy does not justify storing a response in a large RAM, then it might not be worth the cost of storing the response on disk: the extremely low chance that it would be useful in the future might not offset the cost of the disk write or of maintaining relevant meta-data.

4.2. Cooperative caching

Since the hit ratio of a Web cache is a function of the user population, perhaps by combining the efforts of several *cooperative caches* [11], we could improve overall performance. When a proxy P1 receives a request but has no corresponding valid cache entry, P1 could send a request to one or more cooperating caches P2, P3, ... Pn asking if they have a valid entry for the resource. This pays off if the entry is indeed in one of the cooperating caches, and if these caches are close enough so that the inter-cache transfer has a latency or bandwidth advantage over a direct retrieval from the origin server.

One might expect cooperative caching to pay off for two reasons. First, the effective cache size is larger (reducing the probability of a capacity miss). Second, the cooperating caches see a combined reference stream from a larger set of users, which reduces the probability of a compulsory miss; a resource being referenced by a client of P1 might have already been referenced by a client of P2.

Cooperative caching does have its costs. The inter-cache messages add more latency to a reference that ultimately misses in all the caches; these messages and their responses also increase bandwidth demands. If the cooperating caches are arranged in a hierarchy, as is common, the higher levels of the hierarchy may become bottlenecks (imposing queueing delays). Several recent proposals have greatly reduced these costs [19, 51] but the costs cannot entirely be eliminated.

Do the benefits of cooperative caching outweigh the costs? One recent study by Voelker *et al.* [53] suggests that cooperating caching is useful only in limited circumstances. The overall cache hit ratio does increase with increasing population size, but as the population grows above a few thousand users, the additional benefits are very small. Voelker *et al.* state that the population size that can significantly benefit from cooperative caching ‘‘could easily be handled by a single proxy cache.’’ For larger populations, the benefits of cooperative caching appear to be minimal, given current access patterns.

There might be administrative or geographical constraints that prevent the use of a single proxy for a set of small clusters of users, but these same constraints might also work against cooperative caching: separate administrations might have security concerns about shared caches, and between geographically distinct clusters the network latencies and bandwidth might make the inter-cache protocol infeasible.

In any case, a cooperating caching system serving a given set of users cannot provide a better hit ratio than an idealized infinite cache. Since studies have shown that even infinite caches could not provide especially good hit ratios, this suggests that we must look elsewhere for improved Web bandwidth and latency.

5. Squeezing more performance out of HTTP caches

Given the apparent limits on the performance of simple caches, researchers and vendors have developed several techniques to extend the utility of Web caches. If the basic principle of simple ‘‘re-use’’ caching is to exploit repeated references to entire cached responses, the basic principle of these extended mechanisms is to exploit partial information present in caches.

There are at least three such kinds of partial information:

Clues about future references: Because a cache sees an entire reference stream, it can use the information in the reference stream to make predictions about future references. This may allow the cache to more accurately *prefetch* data before it is actually referenced, potentially resulting in much lower latencies. It may also allow the cache to make better replacement decisions.

Filling in or replacing gaps: A cached response might hold some, but not all, of the bits required to satisfy a subsequent request for the same URL. If the cache entry contains missing information, the gap could be filled in using a partial transfer. Or, the underlying resource might have changed, but in such a way that it would be more efficient to transfer the differences between the cache entry and the current resource instance, rather than retrieving an entire new response.

Alias discovery: Quite frequently, the same content appears in the Web under more than one URL: there are multiple *aliases* for a given piece of data. If the cache can detect such aliasing, it might be able to avoid storing multiple copies, or to avoid retrieving data that it already has stored under another alias.

Techniques that exploit partial information typically impose a tradeoff. For example, they may require additional computation, or additional storage; within certain limits, this is usually worthwhile if it reduces network bandwidth or latency. Or, a technique may improve latency at the expense of bandwidth, either by transferring a few additional HTTP header bytes in some cases, or by transferring additional messages. Or, one might be able to improve overall bandwidth by transferring a few additional bytes in carefully chosen cases.

The rest of this paper describes various proposals to exploit partial information in order to increase the utility of HTTP caches. For each proposal, I will try to evaluate the tradeoffs involved.

5.1. Prefetching and replacement hints

If a cache can predict a user's future references, and if there is spare bandwidth, the cache can prefetch the expected resources. When the prediction is correct and timely, this can increase the BHR and reduce user-perceived latency [6, 30, 32], but inevitably increases bandwidth requirements (because of false prefetches). Prefetching forces a cache operator to choose between improving latency and minimizing bandwidth.

Successful prefetching therefore requires solution of two problems: making accurate predictions, and deciding whether sufficient extra bandwidth exists.

Prefetch prediction algorithms use the reference stream, usually on a per-user-session basis, in two ways. First, they observe the stream over relatively long periods in order to construct a model of the conditional probabilities of observing certain references given a set of previous references. Second, they observe the recent behavior of the stream (sometimes just the most recent reference) as input to this model, which may then generate a prediction of some set of future references. For example, if the most recent reference is to `http://ieee.org/index.html`, the model might predict a reference to `http://ieee.org/about/` with probability 0.2, and a reference to `http://ieee.org/conferences/` with probability 0.6.

The first proposals for Web prefetch prediction employed first-order Markov models [5, 45]. Other model-based mechanisms, with better prediction performance or storage requirements, have since been described [46]. Alternatively, a proxy can parse the HTML responses it forwards, extracting the link URLs from these, and then prefetch the link targets [8, 12].

Regardless of the particular prediction algorithm, such a prefetching system cannot predict every reference: many references will have never appeared before in the reference stream, or involve parameters (such as in a POST-based form) that are not visible to a proxy cache. Also, since a prefetch delivers data in advance of the actual reference, a response that cannot be cached (e.g., due to coherency requirements) cannot be prefetched. Kroeger *et al.* report a trace-based study demonstrating that, with infinite cache capacity and infinite bandwidth, at best one can get a 60% latency reduction from any prefetching mechanism that obeys reasonable causality constraints [30]. In other words, at least 40% of the latency in this trace was due to compulsory and coherency misses. Obviously, any real prefetching proxy would be constrained by both cache size and bandwidth.

Prefetching makes sense only if sufficient unused bandwidth exists; otherwise, the extra transfers for unnecessary prefetch would cause network congestion. In a few special cases, one can readily determine that extra bandwidth is available (e.g., when a home user's dialup modem is idle), but in the general case this is a difficult problem, with relatively little research available. Maltzahn *et al.* were able to show that a diurnal variation in demand-fetch bandwidth requirements leaves capacity, during the early morning, for some prefetching, but their work made several simplifying assumptions [34].

Predictions about future references can also be useful in making replacement decisions. Caches typically make such predictions by applying an algorithm, such as LRU, to their own reference streams (for HTTP caches, better algorithms than LRU have been developed [9]). However, because of the Zipf's law distribution of references, such a cache will seldom have sufficiently precise information about most of its entries. A server could supply hints, in the HTTP headers, about the appropriate replacement strategy for a response [13, 36]. Hint information could include both the expected frequency of future references, and the expected cost of a future retrieval.

5.2. Partial transfers, delta encoding, and macros

HTTP transfers can terminate in mid-stream due to network errors, users clicking the ''stop'' button, or when users click on a link before the entire page is loaded. In HTTP/1.0, the result of a partial transfer is not worth caching, but in HTTP/1.1, a cache can fill in the missing data using a ''range retrieval request'' [20]. The ability to retrieve ranges of a response makes partial cache entries useful: although the proxy cannot avoid an additional HTTP request/response round trip, it can avoid transferring bytes that have already been sent over the network. Unfortunately, there are no published statistics for the prevalence or size distribution of partial transfers, but anecdotal evidence suggests they are not uncommon.

The use of range retrievals in HTTP/1.1 imposes a minimal tradeoff, limited to the transmission of one or two extra request header lines. Since the requesting cache knows that it has partial content and wants the full response, the only uncertainty is whether the server is able to generate a partial response.

Partial content as the result of a truncated transfer represents a special case of a more general situation: a cache entry already contains some arbitrary subset of the bits of the desired response. Proposals intended to exploit these bits include delta encoding, cache-based compaction, and HTML macros.

The motivation behind delta encoding [2, 24] is the observation that, although many frequently-referenced Web resources change too frequently to allow for useful caching, the changes are often minor. Instead of sending the entire current instance of a resource, the server sends just the difference (or *delta*) between the cached instance and the current instance. With careful encoding [25], these differences can be quite small. The potential savings vary by Content-type, because much Web content, such as images and continuous media, tend to change more radically than HTML text, but one trace-based study [39] showed potential overall bandwidth savings of 8.5% and latency savings of 5.6%, and significantly greater improvements on text content.

The use of delta encoding in HTTP requires relatively little protocol overhead, and no extra round-trips, but it does require the origin server either to store previous instances of a changing resource, or to be able to dynamically recreate these instances. It also requires a modest amount of computation to create and decode the deltas.

Woo and Chan proposed cache-based compaction [56], which they describe as a further generalization of delta encoding, combined with the use of dictionary-based compression. Dictionary-based compression algorithms, such as Lempel-Ziv-Welch [54], use a dictionary to map between input symbols and compressed codes. In order to decompress the output, the receiver must have the dictionary, which means that it is normally transmitted (in some form) as part of the compressed file.

Whereas delta encoding transmits the difference between the current instance of a resource and one older cached instance of (usually) the same resource, cache-based compaction uses a larger set of cached older instances, possibly from a variety of different resources. This set of older instances functions as a large compression dictionary; the new instance is compressed using this effective dictionary, which does not itself have to be transmitted (because both cache and server are using the same set of older instances).

Cache-based compaction should improve upon the performance of delta encoding (because it includes delta encoding as a special case). However, the protocol is more complex, because the mechanism must choose an appropriate subset of cache entries (it would be infeasible for the cache to inform the server of its entire state), and because the cache and server need to agree on what set of older instances is employed. So far, the proposal has not been extensively evaluated, but it appears to have advantages mostly for low-bandwidth links.

The approaches described previously in this section operate independent of the syntax and semantics of the data being transferred (although delta encoding algorithms for images may require some specialization). They function by decomposing responses at the bit or byte level into currently-cached and need-to-be-transferred components. One can also do this decomposition at a higher level. Douglis *et al.* [15] describe an HTML *macro* mechanism, in which a set of similar HTML pages is decomposed into a constant component (akin to a macro body) and a variable component (akin to macro arguments). In many cases, the variable component can be quite small; this means once the constant component is in a cache, references to similar pages require fetching only the small variable component, at a significant cost savings over transferring a monolithic response.

The main drawback to the HTML macro approach is that it requires direct involvement by the designer (or software) when generating the Web pages, including some careful attention to the decomposition of a set of similar pages. It might also require some additional language-level standardization, although this could perhaps be obviated through the use of Java-based macros.

6. Alias discovery and automatic duplicate suppression

The techniques for extending cache utility discussed in section 5 do not exploit one possible mechanism for exploiting existing cache entries: if two distinct resources would generate exactly identical responses, then a cache entry for one of them could be used to provide a cache hit for a request referencing the other*. This technique is called *duplicate suppression*, and in principle could avoid a lot of data transfer and related latency.

In practice, its utility depends on:

1. A simple and efficient way to detect exact duplication.
2. A simple and efficient HTTP protocol extension to carry the necessary meta-information.
3. A sufficient rate of duplication to justify deploying the protocol extension.

The first requirement is met by the use of a digest (or checksum) algorithm, such as MD5 [48], for which it is difficult to generate identical digest values (‘‘collisions’’) for two different inputs. However, there is some suspicion that MD5 is not collision-proof [49], and if an attacker could generate a carefully chosen collision, this would create a security hole. Other more secure digest algorithms, such as

---

*Cache-based compaction can do this in principle, but only if the two resources have similar URLs; this is often not true, in practice.

SHA-1 [42] or RIPEMD-160 [47], could increase security but at the cost of increased HTTP header sizes and computational costs.

Several protocols have been proposed to address the second requirement. The first such design, the ‘‘Distribution and Replication Protocol’’ (DRP) [23], proposed creating a special Universal Resource Name (URN) out of the MD5 or SHA [42] digest for a resource. A later refinement of this proposal retains the traditional HTTP URL mechanism for naming resources, and transmits the digest in a new HTTP header field [38]. Section 6.2 presents a simplified description of this proposal; it involves sending hint information from servers to clients, and other hint information from clients to caches, in the hope that the caches can avoid requesting duplicated responses directly from the servers.

Finally, we need to evaluate whether the rate of duplication is high enough that it justifies the extra protocol overhead, computational overhead, and implementation complexity of the proposal. Section 6.3 summarizes the results of a trace-based study that addresses this question.

6.1. Related work

Several other techniques also use digests to exploit exact data equality. If duplication is indeed common, then a caching proxy could end up storing multiple copies of many response bodies. By computing a digest of every cachable response body, and maintaining an index keyed by the digest results, the proxy can detect this situation, and arrange to store only one copy of the duplicated body. Inktomi’s Traffic Server [26] uses this technique.

Given that duplication of HTTP bodies is common, one might also expect to see exact duplication at the packet level. Santos and Weatherall [50] describe a router-based technique which detects when a packet body is a duplicate, and sends its digest value instead of the entire body. This mechanism requires the sending router to have fairly accurate information about the state of the receiving router’s packet cache. Santos and Weatherall report bandwidth savings of about 20%, with relatively little overhead. They also report an HTTP-specific packet duplication ratio of about 26%, based on trace analysis.

6.2. Proposed duplicate suppression protocol

This section presents a simplified duplicate suppression extension to HTTP, in order to make the rest of the paper more concrete. Complete specifications for several complete protocols are available [23, 40].

Although users occasionally load Web pages by typing a URL, in most cases an HTTP transfer is initiated when the browser software follows a link. Except in the relatively infrequent case where a link leads to another server, the source of the linkage information is also the source of the linked-to resource. I.e., the same server often controls both the link information and its target.

This allows the server to provide meta-information about the link target as part of the linkage information. (For example, HTML supports the HEIGHT and WIDTH attributes of an IMG tag, allowing the browser to reserve screen space for an image before actually loading it.) To support duplicate suppression, the server includes in this meta-information an MD5 (or similar) digest of the link target. In addition to the possibility of a new HTML attribute, the digest value could be transmitted in a structured type: DRP introduces a new ‘‘index’’ content-type to provide meta-information for a consistent set of link targets [23]; similarly, WEBDAV [22] specifies a similar ‘‘collection’’ resource, whose state consists of a list of member URLs and an extensible set of properties.

Assume then that a client, about to make a request for a URL *U*, has a server-supplied hint that the proper response has an MD5 digest value *D*. The client can check its cache not only for an existing entry for *U*, but also for an existing entry with a digest value of *D*. Either cache entry should therefore be a satisfactory substitute for getting a response from the actual server.

If the client’s local cache does not contain the target, it might send its request via a proxy cache that does. This request would be, in essence, ‘‘please send me either a response for URL *U*, or a response with MD5 digest value *D*''; if the proxy cache has a response cached under either key, it can return the cache hit rather than forwarding the request to the server. Thus, once the client knows the proper MD5 digest value, it can use both its own cache and a proxy's cache to find a duplicate with the same digest, rather than waiting for a response from the actual server.

The complete specification of a duplicate suppression protocol would require attention to a number of other issues, such as whether HTTP header information for a cached response (such as authentication information) can properly be associated with a duplicate-suppression response for a different resource, and whether the cache entries discovered by duplicate suppression hints are timely with respect to the requested URL.

6.3. Potential benefits of duplicate suppression

Duplicate suppression can be evaluated by simulating its effect on a reference stream trace, such as one taken from a proxy server; the trace would have to include the digest value of every response body. However, without actual deployment of a duplicate suppression protocol, one cannot know often clients would actually receive server-supplied digest hints. For this reason, a trace-based simulation of current references can provide only an upper bound on the potential improvements.

I obtained a 23-day trace from a non-caching proxy at Compaq Computer Corporation. The raw trace includes 29 million entries, although only about 19 million are useful for this study. The trace was then fed to a simulator which modeled an infinite cache; this avoids any capacity misses, and so reduces the apparent advantage of duplicate suppression. The simulation also models a ''perfect coherency'' cache, since the MD5 digests allow the simulator to know for sure whether a coherency miss would be necessary or not.

Space permits only a brief summary of the simulation results; more detail is available in [37].

The infinite perfect-coherency cache simulation, without duplicate suppression, yielded a hit ratio of 50.4%, and a BHR of 32.2%. Duplicate-suppression ''hits'' always take the place of cache misses, not cache hits, so their benefit always adds to that of simple caching. Had the duplicate-suppression mechanism been applied to every eligible request, it would have avoided an additional 5.4% of the retrievals in the trace. Weighted by bytes transferred, duplicate suppression would have saved an additional 6.2% of the server response bytes.

Figure 6-1 shows the unweighted cache-hit ratio and duplicate-suppression ratios, sampled at 24-hour intervals. The curves trend upward because the simulations start with an empty (''cold'') cache. The figure also shows the unweighted ''total'' ratio (including both cache hits and duplicate-suppression ''hits''). Figure 6-2 shows the corresponding byte-weighted ratios.

The weighted cache-hit ratio is much lower than the unweighted cache-hit ratio, but the weighted duplicate-suppression ratio is actually higher than the unweighted duplicate-suppression ratio. Therefore, the net bandwidth improvement due to duplicate suppression is more significant than one would expect from the unweighted ratios.

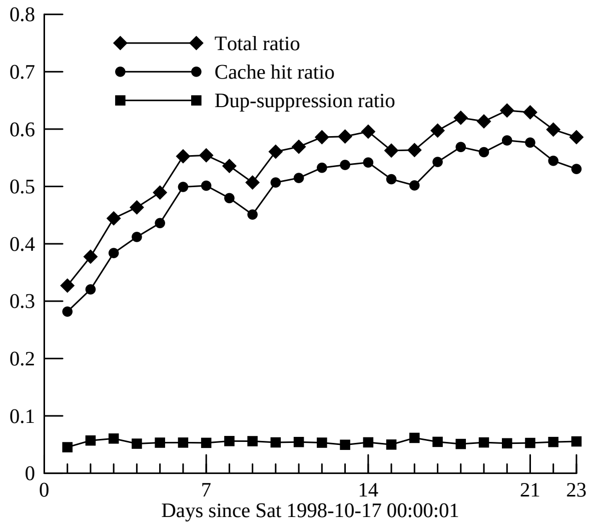

Figure 6-1: Daily ratios

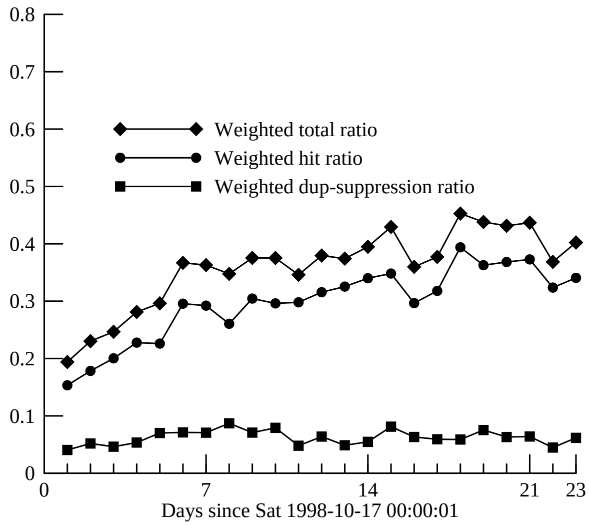

Figure 6-2: Daily ratios, weighted by response size in bytes

The unweighted duplication ratio never gets much above 6%, on a daily basis, which implies that it might not be worth the extra protocol overhead. This statistic, however, is calculated over all responses in the trace, but one might expect that some sets of resources are far more likely than others to be subject to duplication. If so, one could limit the protocol overhead of duplicate suppression to these contexts, and concentrate the effort where the benefits justify the costs. For example, ‘‘audio/midi’’ content (relatively rare in this trace) yields a duplication ratio of 15%, and a weighted ratio of 15.5%. Other content-types with relatively high ratios include Java byte codes, other program binaries, GIF and JPEG images, and some video formats.

Almost 84% of the URLs in the trace were never involved in duplication; 16% were duplicated exactly once, and 0.15% were duplicated exactly twice. In other words, very few URLs are duplicated more than once, but some highly-duplicated URLs account for most of the duplication. In fact, half of all duplicate responses come from URLs that give rise to at least 406 different duplicate responses. Also, most duplication is limited to a very small subset of the server hosts in the trace.

These results and others suggest that one could predict, based on content-type, server host, or other factors, whether a response is likely to experience duplication. This would allow the server to avoid sending digest hints except in cases where they would probably pay off.

7. Summary

This paper has surveyed a number of techniques for better exploiting the bits in HTTP caches. How do these techniques compare? Table 7-1 summarizes the tradeoffs for each technique, on five metrics: latency, bandwidth requirements, cache coherency, CPU time (on servers, proxies, and clients), and proxy RAM requirements. The last column lists the contexts where each approach appears to be most applicable. The table entries are guesses, at best, pending more extensive studies.

Note that most of the entries in table 7-1 do not affect cache coherency. Several techniques to improve HTTP cache coherency have been proposed [10, 29]. Because current methods for avoiding incoherency often disable caching, these new techniques may provide improvements on other metrics. Space limitations prevent further discussion here.

Technique	Latency	Bandwidth	Coherency	CPU	RAM	Applicability
Prefetching	++	– –	+/–	–	–	General
Replacement hints	+	+?	=	=	+	General
Partial transfers	+	++	=	=	–	General
Delta encoding	+	+	=	–	– –	Mostly HTML, text
Cache-based compaction	+?	+?	=	–	=?	Mostly HTML, text
HTML macros	–	+	=	=	=	HTML only
Duplicate suppression	+	+	=	–	– –	Image, stream, and program objects; maybe HTML and text

Symbol	Meaning
++	Much improved
+	Somewhat improved
=	Not significantly affected
+/–	Depends on circumstances
–	Somewhat worse
– –	Much worse
?	Needs more study

Table 7-1: Comparison of HTTP caching techniques

8. Conclusions

HTTP caching remains a fertile area both for research and for development of commercial products and services. Web caching researchers initially focussed on replacement policies and cooperative caching, but these lines of research may be reaching diminishing returns. The wide variety of recent proposals to improve the effectiveness of Web caches suggests that the space of possible solutions has not yet been fully explored. Even for the techniques described in this paper, we still lack sufficient understanding of their utility, and of how best to implement and deploy them.

Acknowledgments

I would like to thank Mark Manasse and Andrei Broder for their help in explaining to me various aspects of cryptography and digital signatures.

References

[1] Jussara Almeida and Pei Cao. Measuring Proxy Performance with the Wisconsin Proxy Benchmark. In *Proc. 3rd International WWW Caching Workshop*. Manchester, England, June, 1998.

[2] G. Banga, F. Douglis, and M. Rabinovich. Optimistic Deltas for WWW Latency Reduction. In *Proc. 1997 USENIX Technical Conf.*, pages 289-303. Anaheim, CA, Jan., 1997.

[3] Paul Barford, Azer Bestavros, Adam Bradley, and Mark E. Crovella. Changes in Web Client Access Patterns: Characteristics and Caching Implications. *World Wide Web, Special Issue on Characterization and Performance Evaluation* 2(1):15-28, January, 1999.

[4] Paul Barford and Mark Crovella. Generating representative Web workloads for network and server performance evaluation. In *Proc. Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '98/PERFORMANCE '98)*, pages 151-160. Madison, WI, June, 1998.

[5] Azer Bestavros. Using Speculation to Reduce Server Load and Service Time on the WWW. In *Proc. 4th International Conference on Information and Knowledge Management*. Baltimore, MD, November, 1995.

[6] A. Bestavros and C. Cunha. *A Prefetching Protocol Using Client Speculation for the WWW*. Technical Report TR-95-011, Boston University, CS Dept, Boston, MA, Apr., 1995.

[7] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proc. Infocom '99*, pages ??-??. New York, NY, March, 1999.

[8] CacheFlow, Inc. XXX former press release? http://www.cacheflow.com.

[9] Pei Cao and Sandy Irani. Cost-Aware WWW Proxy Caching Algorithms. In *Proc. USENIX Symp. on Internet Technologies and Systems*, pages 193-206. Monterey, CA, December, 1997.

[10] Pei Cao, Jin Zhang, and Kevin Beach. Active Cache: Caching Dynamic Contents on the Web. In *Proc. Middleware '98*, pages 373-388. The Lake District, England, September, 1998.

[11] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A Hierarchical Internet Object Cache. In *Proc. 1996 USENIX Technical Conference*, pages 153-163. San Diego, CA, January, 1996.

[12] Ken-ichi Chinen and Suguru Yamaguchi. An Interactive Prefetching Proxy for Improvement of WWW Latency. In *Proc 7th Annual Conf. of the Internet Society*. Kuala Lumpur, June, 1997.

[13] E. Cohen, B. Krishnamurthy, and J. Rexford. Evaluating Server-Assisted Cache Replacement in the Web. In *Proc. European Symposium on Algorithms*, pages 307-319. Venice, Italy, August, 1998.

[14] F. Douglis, A. Feldmann, B. Krishnamurthy, and J. Mogul. Rate of Change and Other Metrics: a Live Study of the World Wide Web. In *Proc. Symp. on Internet Technologies and Systems*, pages 147-158. USENIX, Monterey, CA, Dec., 1997.

[15] F. Douglis, A. Haro, and M. Rabinovich. HPP: HTML Macro-Preprocessing to Support Dynamic Document Caching. In *Proc. Symp. on Internet Technologies and Systems*, pages 83-94. USENIX, Monterey, CA, Dec., 1997.

[16] B. Duska, D. Marwood, and M. Feeley. The Measured Access Characteristics of World-Wide-Web Proxy Caches. In *Proc. Symp. on Internet Technologies and Systems*, pages 23-35. USENIX, Monterey, CA, Dec., 1997.

[17] Equilibrium. DeBabelizer Product Information page. http://www.equilibrium.com/ProductInfo/ProdInfo.html.

[18] J. Ernst, C. Fraser, W. Evans, S. Lucco, and T. Proebsting. Code compression. In *Proc. ACM SIGPLAN '97 Conf. on Programming Language Design and Implementation*, pages 358-365. Las Vegas, NV, June, 1997.

[19] L. Fan, P. Cao, J. Almeida and A. Broder. Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol. In *Proc. SIGCOMM '98*, pages 254-265. Vancouver, Sept., 1998.

[20] R. Fielding, J. Gettys, J. Mogul, H. Nielsen, and T. Berners-Lee. *Hypertext Transfer Protocol -- HTTP/1.1*. RFC 2616, HTTP Working Group, Jun., 1999.

[21] A. Fox, S. Gribble, E. Brewer, and E. Amir. Adapting to Network and Client Variation via On-Demand Dynamic Transcoding. In *Proc. ASPLOS VII*, pages 160-170. Cambridge, MA, Oct., 1996.

[22] Y. Goland, E. Whitehead, Jr., A. Faizi, S. Carter, D. Jensen. *HTTP Extensions for Distributed Authoring -- WEBDAV*. RFC 2518, IETF, Feb., 1999.

[23] A. van Hoff, J. Giannandrea, M. Hapner, S. Carter, and M. Medin. *The HTTP Distribution and Replication Protocol*. Technical Report NOTE-DRP, World Wide Web Consortium, Aug., 1997. http://www.w3.org/TR/NOTE-drp-19970825.html.

[24] B. Housel and D. Lindquist. WebExpress: A System for Optimizing Web Browsing in a Wireless Environment. In *Proc. 2nd Annual Intl. Conf. on Mobile Computing and Networking*, pages 108-116. Rye, New York, Nov., 1996.

[25] J. Hunt, K. P. Vo, and W. Tichy. An Empirical Study of Delta Algorithms. In *IEEE Soft. Config. and Maint. Workshop*. Berlin, Mar., 1996.

[26] Inktomi Corporation. Inktomi Traffic Server product info. URL http://www.inktomi.com/products/network/traffic/product.html. 1999

[27] ISO/IEC. *11172-3, Information technology -- Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s -- Part 3: Audio* 1993.

[28] James J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proc. 13th ACM Symposium on Operating Systems Principles*, pages 213-25. October, 1991.

[29] Balachander Krishnamurthy and Craig E. Wills. Study of Piggyback Cache Validation for Proxy Caches in the World Wide Web. In *Proc. USENIX Symp. on Internet Technologies and Systems*, pages 1-12. Monterey, CA, December, 1997.

[30] T. Kroeger, D. Long, and J. Mogul. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *Proc. Symp. on Internet Technologies and Systems*, pages 13-22. USENIX, Monterey, CA, Dec., 1997.

[31] H. Lie and B. Bos. *Cascading Style Sheets, level 1*. Recommendation REC-CSS1, World Wide Web Consortium, Dec., 1996. http://www.w3.org/pub/WWW/TR/REC-CSS1.

[32] T. Loon and V. Bharghavan. Alleviating the Latency and Bandwidth Problems in WWW Browsing. In *Proc. Symp. on Internet Technologies and Systems*, pages 219-230. USENIX, Monterey, CA, Dec., 1997.

[33] Carlos Maltzahn, Kathy J. Richardson, and Dirk Grunwald. Reducing the Disk I/O of Web Proxy Server Caches. In *Proc. 1999 USENIX Annual Technical Conference*, pages 225-238. Monterey, CA, June, 1999.

[34] Carlos Maltzahn, Kathy J. Richardson, Dirk Grunwald, and JamesMartin. On Bandwidth Smoothing. In *Proc. 4th International Web Caching Workshop*. San Diego, CA, March, 1999.

[35] Evangelos P. Markatos, Manolis G.H. Katevenis, Dionisis Pnevmatikatos, and Michail Flouris. Secondary Storage Management for Web Proxies. In *Proc. Second USENIX Symp. on Internet Technologies and Systems*, pages XXX-XXX. Boulder, CO, October, 1999.

[36] Jeffrey C. Mogul. Hinted caching in the Web. In *Proc. Seventh ACM SIGOPS European Workshop*, pages 103-108. Connemara, Ireland, September, 1996.

[37] Jeffrey C. Mogul. A trace-based analysis of duplicate suppression in HTTP. 1999.In preparation.

[38] J. Mogul, B. Krishnamurthy, F. Douglis, A. Feldmann, Y. Goland, and A. van Hoff. *Delta encoding in HTTP*. Internet-Draft draft-mogul-http-delta-00, IETF, January, 1998. This is a work in progress.

[39] J. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *Proc. SIGCOMM '97*, pages 181-194. Cannes, France, Sept., 1997.

[40] J. Mogul and A. van Hoff. *Duplicate suppression in HTTP*. Internet-Draft draft-mogul-http-dupsup-00, IETF, Apr., 1998. This is a work in progress.

[41] L. B. Mummert, M. R. Ebling, and M. Satyanarayanan. Exploiting Weak Connectivity for Mobile File Access. In *Proc. 15th Symposium on Operating Systems Principles*, pages 143-155. Copper Mountain, CO, December, 1995.

[42] National Institute of Standards and Technology. *Secure Hash Standard*. FIPS Pub. 180-1, U.S. Department of Commerce, Apr., 1995. http://csrc.nist.gov/fips/fip180-1.txt.

[43] National Laboratory for Applied Network Research (NLANR). NLANR Hierarchical Caching System Usage Statistics. http://www.ircache.net/Cache/Statistics/.

[44] National Laboratory for Applied Network Research (NLANR). Cache Popularity Index. http://www.ircache.net/Cache/Statistics/
Popularity-Index/.

[45] V. Padmanabhan and J. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. *Computer Communication Review* 26(3):22-36, 1996.

[46] Jim Pitkow and Peter Pirolli. Mining Longest Repeated Subsequences to Predict World Wide Web Surfing. In *Proc. Second USENIX Symp. on Internet Technologies and Systems*, pages XXX-XXX. Boulder, CO, October, 1999.

[47] B. Preneel, A. Bosselaers, and H. Dobbertin. The cryptographic hash function RIPEMD-160. *CryptoBytes* 3(2):9-14, Autumn, 1997.

[48] R. Rivest. *The MD5 Message-Digest Algorithm*. RFC 1321, Network Working Group, April, 1992.

[49] M. Robshaw. On Recent Results for MD2, MD4 and MD5. *RSA Laboratories' Bulletin* 4(12):1-6, Nov. 12, 1996.

[50] J. Santos and D. Wetherall. Increasing Effective Link Bandwidth by Suppressing Replicated Data. In *Proc. USENIX 1998 Annual Technical Conf.*, pages 213-224. New Orleans, LA, June, 1998.

[51] Renu Tewari, Michael Dahlin, Harrick M. Vin, Jonathan S. Kay. Design Considerations for Distributed Caching on the Internet. In *Proc. 19th IEEE Int'l Conf. on Distributed Computing Systems*, pages XXX-XXX. Austin, TX, May, 1999.

[52] K. Thompson, G. J. Miller, and R. Wilder. Wide-Area Internet Traffic Patterns and Characteristics. *IEEE Network* 11(6):10-23, Nov./Dec., 1997.

[53] Geoff Voelker, Alec Wolman, Nitin Sharma, Neal Cardwell, Anna Karlin, and Hank Levy. On the Scale and Performance of Cooperative Web Proxy Caching. In *Proc. 17th SOSP*, pages XXX-XXX. Kiawah Island, SC, December, 1999.

[54] T. Welch. A Technique for High Performance Data Compression. *IEEE Computer* 17(6):8-19, 1984.

[55] S. Williams, M. Abrams, C. Standridge, G. Abdulla, and E. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *Proc. SIGCOMM '96*, pages 293-305. Stanford, CA, Aug., 1996.

[56] Mun Choon Chan and Thomas Woo. Cache-based Compaction: A New Technique for Optimizing Web Transfer. In *Proc. IEEE Infocom '99*, pages 117-125. New York, NY, March, 1999.

[57] T. L. Yu. Data Compression for PC Software Distribution. *Software - Practice and Experience* 26(11):1181-1195, 1996.

[58] DUMMY ENTRY. FIX THIS REFERENCE BEFORE PUBLICATION. .