Enhancement and Validation of Squid's Cache Replacement Policy

John Dilley
Martin Arlitt
Stéphane Perret

HP Laboratories, Palo Alto, CA

Abstract

Caching in the World Wide Web has been used to enhance the scalability and performance of user access to popular web content. Caches reduce bandwidth demand, improve response times for popular objects, and help reduce the effects of so called "flash crowds". There are several cache implementations available from software and appliance vendors as well as the Squid open source cache software.

The cache replacement policy decides which objects will remain in cache and which are evicted to make space for new objects. The choice of this policy has an effect on the network bandwidth demand and object hit rate of the cache (which is related to page load time). This paper reports on the implementation and characterization of two newly proposed cache replacement policies in the Squid cache.

1. Introduction

In the World Wide Web, caches store copies of previously retrieved web objects to avoid transferring those objects upon subsequent request. By preventing future transfer, the cache reduces the network bandwidth demand on the external network, and usually reduces the average time it takes for a web page to load.

Web caches are located at several points through the Internet, as depicted in Figure 1.

- At the client browser, both in memory and on disk for persistent storage across sessions.
- In a local proxy cache server, typically on the same network as the client browser. These proxy servers act on behalf of the client to communicate across the external network. Such caches may be arranged in a hierarchy and communicate with enterprise or backbone caches.
- At the origin server site, where the cache is referred to a reverse proxy server. The primary goal of a reverse proxy server is to reduce server load as opposed to network bandwidth or user latency.

Proxy caches can be implemented either as explicit or transparent proxies. Explicit proxies require the user to configure their browser to send HTTP GET requests to the proxy. Transparent proxies do not require the browser to be explicitly configured; instead, a network element (switch or router) in the connection path intercepts requests to TCP port 80 and redirects that traffic to the cache. The cache then determines if it can serve the object at all, and if so whether the object is already in cache. Since cached objects can change on the origin server without the cache being informed, a proxy cache identifies whether objects in its cache are fresh or stale based upon the object's last modification time and the time of last retrieval or validation. If the object is fresh it is served immediately from the cache. If it is determined to be stale the cache makes a validation request of the origin server to determine if the object has changed since the last retrieval. The validation returns a fresh copy of the object or a status code indicating the object has not changed.

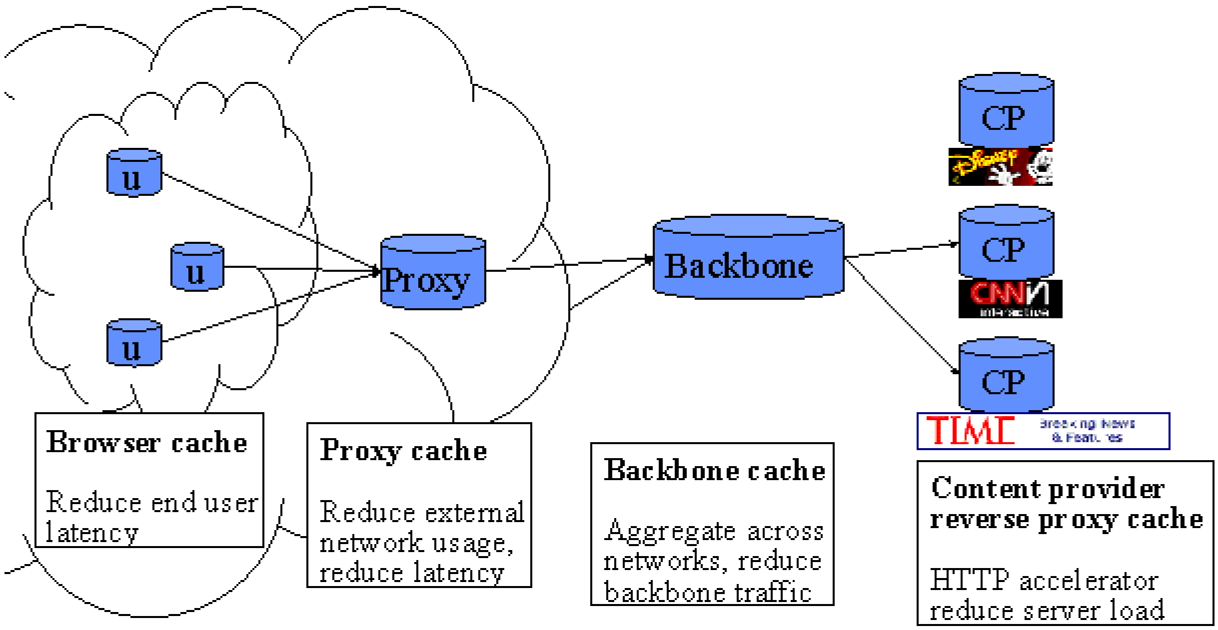


Figure 1: Web Cache Architecture

1.1 The Role of the Cache Replacement Policy

A cache server has a fixed amount of storage for holding objects. When this storage space fills up the cache must choose one or more objects to evict in order to make room for (newly referenced) objects. The cache replacement policy determines which objects should be removed from the cache. The goal of the replacement policy is to make the best use of available resources, including disk and memory space and network bandwidth. Since web use is the dominant cause of network backbone traffic today [TMW97], the choice of cache replacement policies can have a significant impact in global network traffic, as well as local resource utilization.

1.1.1 Web Proxy Workload Characterization

A cache replacement policy must be evaluated with respect to an offered workload. The workload describes the characteristics of the requests being made of the cache. Of particular interest is the pattern of references: how many objects are referenced, and what is the relationship among accesses. Typically workloads are sufficiently complicated that they cannot be described with a simple formula. Instead, traces of actual live execution are often the best way to describe a realistic workload. This has the advantage of being real (as compared with a synthetic workload), but has the drawback of not capturing changing (especially future) behavior, or behavior of a different set of users. Once a workload is available, either analytical or empirical, the efficiency of various cache implementations can be compared.

Recent studies of web workloads have shown tremendous breadth and turnover in the popular object set [AFJ99][DFK97]. In [AFJ99] we describe in detail the characterization of a large data set obtained by tracing every request made by a population of thousands of home users connected to the web via cable modem technology over a five month period. With this trace it is possible to observe long-term dynamics in request traces of real users. Arlitt et al developed a simulator that could explore the behavior of various cache replacement policies under this empirical workload. This workload characterization led to the development of new cache replacement policies, which inspired the Squid implementation and validation described herein.

1.1.2 Measures of Efficiency

The most popular measure of cache efficiency is hit rate. This is the number of times that objects in the cache are re-referenced. A hit rate of 70% indicates that seven of every 10 requests to the cache found the object being requested.

Another important measure of web cache efficiency is byte hit rate. This is the number of bytes returned directly from the cache as a fraction of the total bytes accessed. This measure is not often used in classical cache studies because the objects (cache lines) are of constant size. However, web objects vary greatly in size, from a few bytes to millions. Byte hit rate is of particular interest because the external network bandwidth is a limited resource (sometimes scarce, often expensive). A byte hit rate of 30% indicates that 3 of 10 bytes requested by clients were returned from the cache; and conversely 70% of all bytes returned to users were retrieved across the external network link.

Other measures of cache efficiency include the cache server CPU or IO system utilization, which are driven by the cache server's implementation. Average object retrieval latency (or page load time) is a measure of interest to end users and others. Latency is inversely proportional to object hit rate -- a cache hit served without remote communication is quicker to complete than a request that must pass through the cache to an origin server and back. However, it is not possible to guarantee that latency is minimized by increasing hit rate. It may be that caching a few documents with high download latency would reduce average latency more than caching many low latency documents. End user latency is difficult to measure at the cache, and can be significantly affected by factors outside the cache. Our study focused primarily on (object) hit rate and byte hit rate. We also examine CPU utilization to assess whether the new replacement policies are viable for use in a large, busy cache.

Note that object hit rate and byte hit rate actually trade off against each other. In order to maximize object hit rate it is advantageous to keep many small popular objects. A single large object, say 10 MB, will displace many smaller objects (1024 10 KB objects). However, to optimize the byte hit rate it is better to keep some large popular objects. It is clearly preferable to keep objects that will be popular in the future and evict unpopular ones; the tradeoff is whether to bias against large objects or not. To summarize, an ideal cache replacement policy should be able to accurately determine future popularity of documents and choose how to use its limited space in the most advantageous way. In the real world, we must develop heuristics to approximate this ideal behavior.

1.2 Related Work

Cache replacement has been described and analyzed since processor memory caching was invented. In each case, the combination of replacement policy and offered workload determine the efficiency of the cache in optimizing the utilization of system resources. One of the most popular replacement policies is the Least Recently Used (LRU) policy, which evicts the object that has not been accessed for the longest time. This policy works well when there is a high temporal locality of reference in the workload (that is, when most recently referenced objects are most likely to be referenced again in the near future). The LRU policy is often implemented using a linked list ordered by last access time. Addition and removal of objects from the list is done in O(1) (constant) time by accessing only the tail of the list. Updates when an object is referenced can be accomplished in constant time by keeping a pointer into a list node in object metadata.

Another common policy is Least Frequently Used (LFU). With each reference to an object, a reference count is incremented by one. The LFU policy evicts the objects with the lowest reference count when it makes replacement decisions. Unlike LRU, the LFU policy cannot be implemented as single a linked list (either removal or insertion in the list upon update would take O(N) (linear) time). Sometimes a priority queue (heap) is used to implement the LFU policy. With a heap, insertion and update are done on O(log N) time. Unfortunately, the LFU policy can suffer from cache pollution: if a popular object becomes unpopular, it will remain in the cache a long time, preventing other newly (or slightly) popular objects from replacing it. A variant of the LFU policy, the LFU-Aging policy considers both the access frequency of an object and the age of the object in cache (the recency of last access). The aging policy addresses the cache pollution that occurs due to turnover in the popular object set.

Other research has defined additional replacement policies optimized for the web. The GreedyDual-Size (GDS) policy takes into account size and a cost function for retrieving objects. The GDS policy is proposed in [CI97], which also provides a good survey of existing replacement policies. GDS is explored as well in [AFJ99], where the alternative replacement policies we explore are proposed.

In [ACDFJ99] the GDS policy is refined to account for frequency, leading to the definition of a new policy, GDS-F. In [C98] Cherkasova presents a more formal treatment of the replacement policies than provided here and in particular motivates the use of frequency as a replacement policy parameter.

2. Replacement Policy Implementation

In the web, the most prevalent cache server seems to be the Squid public domain freeware [Squid]. The replacement policy used by Squid is the LRU policy. LRU works well in practice and it has been examined in papers at previous Web Caching Workshops [WCW98]. Squid source code is free and has excellent technical assistance through the `squid-users@ircache.net` mailing list, which led to our choice of Squid as a platform to experiment with cache replacement policies.

We designed and implemented in Squid the following variants of the LFU and GDS policies, based upon analysis and trace-based simulation of the web workload mentioned earlier [ACDFJ99].

- LFU with Dynamic Aging (LFUDA): A variant of LFU that uses a dynamic aging policy to accommodate shifts in the set of popular objects. In the dynamic aging policy, the *cache age factor* is added to the reference count when an object is added to the cache or an existing object is modified. This prevents previously popular documents from polluting the cache. Instead of adjusting all key values in the cache, the dynamic aging policy increments the cache age when evicting objects from the cache [AFJ99], setting it to the key value of the evicted object. This has the property that the cache age is less than or equal to the minimum key value in the cache. This also prevents the need for parameterization, which LFU-Aging requires.
- GDS-Frequency (GDSF): A variant of the Greedy Dual-Size policy that takes into account frequency of reference. This policy is optimized for more popular, smaller objects in order to maximize object hit rate. The GDSF policy assigns a key to each object computed as the object's reference count divided by its size, plus the cache age factor. By adding the cache age factor we limit the influence of previously popular documents, as described above for LFUDA.

We also implemented and tested a variant of GDSF that did not include the aging factor. Since the SPECWeb workload does not exhibit turnover in the set of popular objects, these replacement policies behaved identically in our benchmark. However in real usage frequency based policies can suffer significant cache pollution. This underscores the need for caution when interpreting benchmark results.

2.1 Implementation Experiences - Squid 1

The first version of Squid we experimented with was Squid 1.1.20. In this implementation, the cache replacement policy is implemented by occasionally scanning a set of hash buckets and sorting the contents by their last reference time using `qsort()`. This approach is only approximate; it can never be precise since the scan only considers a subset of the buckets each time. However, it is very easy to implement an alternative cache replacement policy -- all that is required is to supply a different `qsort()` comparison function. Fortunately the cache metadata object passed to the comparison function contain all the information needed to execute the replacement policies we were examining. Squid 1 first attempts to remove expired objects (those whose "Expires:" time has passed), but in our benchmark run there was not time for there to be expired objects. In a live workload this would have an impact, however its impact is likely negative since expired objects may not have been modified at the source. For such objects, Squid would send a `GET If-Modified-Since` request that would result in a positive validation (HTTP response code `304 Not Modified`) from the origin server, resulting in significant savings in byte traffic.

We tried a variety of functions to implement the proposed replacement policies. Implementing a simple LFUDA was trivial -- the comparison function required by this policy is simply the reference count added to the cache age factor. Since the Squid 1 replacement policy is only approximate, this property can be violated. In addition, since most objects in the cache have a reference count of one this policy was not very precise. The heap-based Squid 2 implementation described below avoids this problem by adding the heap age to the reference count when inserting an object into the heap to determine the key on which to sort the objects. Therefore, the heap implementation tends to favor the least recently used among these least frequently used objects. This is exactly the point of aging.

Since the cache metadata values of interest were all integral (object size in bytes, reference count), when implementing GDSF we first tried shifting the size by the reference count. This magnified the effect of the reference count relative to the size (equivalent to size / $2^{refcount}$). This operation is highly efficient as compared with floating point division, and since the `qsort()` comparison function is invoked O(N log N) times on average this seemed to be a good idea. We also implemented the full GDSF replacement policy using floating point division. When we characterized its CPU performance we were surprised to find that the CPU utilization did not increase as much as we had expected. In the end we used the original definition with floating point division for our analysis.

2.2 Implementation Experiences - Squid 2

During our initial exploration in the summer of 1998 we also looked at the Squid 1.2 source, which was in beta at that time. We chose to use the more stable Squid 1 source base but observed that in the Squid 1.2.beta code the LRU policy was implemented with a doubly linked list rather than `qsort()` bucket scanning. This is a more sensible and simple way to implement the LRU policy, however it would make our replacement policy implementation more difficult.

We were not entirely happy with the implementation experiences from Squid 1, given that it was only approximate and we felt the replacement policies were not achieving their maximum effectiveness. So towards the end of 1998 we obtained a current version of the Squid source, squid-2.1.PATCH2 and began implementing the GDSF and LFUDA policies for Squid 2. (Note: the release line that was originally called "Squid 1.2" was renamed to "Squid 2.1".)

The first step was to add our metadata heap implementation (`heap.c` and `heap.h`) to the Squid source tree and to modify the cache metadata to contain a heap_node pointer instead of a link_node pointer. This pointer is used to find the object in the heap (or LRU list) after a hash table lookup. Next we found the places where the LRU linked list insertion and deletion calls (`dlinkAdd()`, `dlinkDelete()`) were made in `store.c`, and replaced these with a heap call to insert an object, remove an object, or update the position of an object in the heap. In many cases a `dlinkDelete()` call was immediately followed by a `dlinkAdd()` to move an object to the head of the LRU list (eviction proceeds from the tail of the list). We modified these call pairs to be a heap update operation, which adjusts the position of a node in the heap after its key value is modified. In addition, `store_dir.c` used the LRU list to iterate through all object metadata in order to write the swap log file. We replaced that to iterate through the heap instead.

The replacement policy itself was implemented as the key generation function for the heap. The key generation function is called for each object when it is being added to or updated in the heap -- it computes a key value based upon the object's metadata, depending upon the policy. This value is stored as a floating point value in a heap node (the corollary to the `link_node` structure in Squid's LRU implementation). We added a few lines of code to update each object in the heap wherever cache metadata was modified following a change in document size (in `store_swapout.c` after reading the object from the origin server) or an increment to the reference count following a document access (cache access in `client_side.c`).

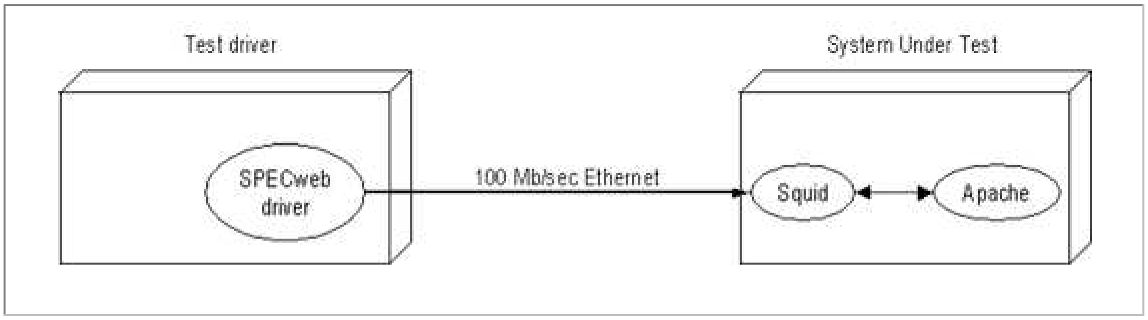
One challenge we faced was what to do with entries that were in use (`storeEntryLocked()` returned true). In the LRU implementation, objects were being put at the head of the list to get them out of the way for a while. In a heap, one cannot simply put them back, as they would be immediately at the top of the heap, resulting in an infinite loop. The effect of moving an object to the head of the LRU list is similar to that of incrementing its reference count and moving it in the heap. But incrementing the reference count is not the right thing to do since first it would not guarantee to move the object; furthermore the reference count would then be incorrect. Instead, we chose to put these objects on a temporary list and reinsert them into the heap once replacement completed. We also noted that certain "special" objects in the cache always appear locked (internal objects representing icons the cache serves up when viewing a directory). In our tests these objects were never used, so always had a reference count of zero and were ideal candidates for eviction from the cache. Instead of moving them to the temporary list and back every time we run the cache replacement policy we chose to remove them from the heap. They remain in cache but are not candidates for replacement.

We also tuned the function that evicts objects from disk, `storeMaintainSwapSpace`. In the original Squid2 implementation this function runs fairly often and often does little work. It makes a complicated computation for each object being considered requiring one floating point multiply, two divides, an exponentiation operation and two branches in order to maintain cache disk (swap) space between a high and low water mark. The calculation is based upon the LRU reference age configuration parameter, which since we are not using LRU does not make sense. Instead, we chose not to call this function (`storeExpiredReferenceAge`) when replacing objects; instead we maintain space between the high and low water marks with a simpler mechanism that evicts objects regardless of their age, considering only the heap key (i.e., strictly using the replacement policy). We replace objects until we reach the existing Squid `max_scan` or `max_remove` limits, which prevent `storeMaintainSwapSpace` from taking too much time when evicting objects, or the low water mark.

Since we were changing some rather significant operational behaviors we also decided to implement an LRU replacement policy using our heap implementation and all of the other modifications to isolate the effect of the changes we made in Squid2. The heap-based LRU implementation is identical to the GDSF and LFUDA implementations except for the replacement policy (key generation function). When compared with GDSF or LFUDA this policy shows the effect of the replacement policy independent of the other changes. When compared with the original LRU policy this policy shows the effect of the other changes we made in Squid in order to implement the heap-based replacement policy. This is explained in further detail in Section 3.2, Squid 2 Validation.

3. Replacement Policy Validation Results

When developing the replacement policies we studied web workloads to identify the characteristics that best identify the valuable objects to keep in the cache. Based upon this we proposed new, parameterless replacement policies. We then implemented the policies in a simulator and ran the five-month trace on each replacement policy to measure its effectiveness in terms of hit rate and byte hit rate. We also wanted to validate our simulation results through empirical measurement to assess how they performed in practice and to assess factors we could not observe in the simulator, such as CPU resource consumption and access latency.



HP K210 with HP-UX 10.20 3 CPU, 1024 MB RAM	**100 MB/sec Ethernet**	**HP K460 with HP-UX 10.20 1 CPU, 512 MB RAM**

Figure 2: Testbed Configuration

In order to validate the cache replacement policies we built both an unmodified Squid cache server and versions that used each of our proposed policies. We configured the Squid server as an HTTP accelerator for an Apache web server on the same node, as depicted in Figure 2. In accelerator mode, the Squid server acts as a reverse proxy cache: it accepts client requests, serves them out of cache if possible, or requests them from the origin server for which it is the reverse proxy. This setup allowed us to use the SPECWeb benchmark package as a workload generator.

SPECWeb was designed to web servers performance, and is not an ideal tool for characterization of proxy performance. We had to modify the SPECWeb client driver to cause it to use a working set of sufficient breadth at lower demand levels. The standard SPECWeb benchmark working set [SPECWeb] is proportional to the target load (ops/sec), but for a proxy benchmark we wanted the broadest possible workload to exercise the replacement policies. We modified the manager script so that client processes always used the maximum working set size regardless of the target throughput. We were also able to create a fileset on disk with 36,000 unique objects in only a fraction of the 2 GB disk space normally required through use of directory symbolic links. This setup was suitable for examining the hit rate of a Squid reverse proxy cache under various replacement policies.

To test the implementations we ran the SPECWeb client driver on a test driver node and pointed it at the Squid HTTP accelerator port on the system under test. We wanted to assess the correctness of the cache implementation as well as its efficiency in terms of hit rate and byte hit rate. We examined the resulting Squid cache logs to ensure that it was replacing documents according to the intent of the replacement policy, as well as to determine the hit rate and byte hit rate of the entire SPECWeb run (there was not a separate warmup period).

3.1 Squid 1 Validation

Figures 3 and 4 illustrate the performance of the Squid 1 implementation under the three replacement policies. The cache server was configured to use the Squid default values of 8 MB RAM and 100 MB of disk in order to stress the replacement policies as much as possible with the SPECWeb workload. We also used the Squid default values for high and low water marks which control when the replacement policy is active (we used Squid defaults everywhere possible, only specifying our own cache directory and port assignment for accelerator mode).

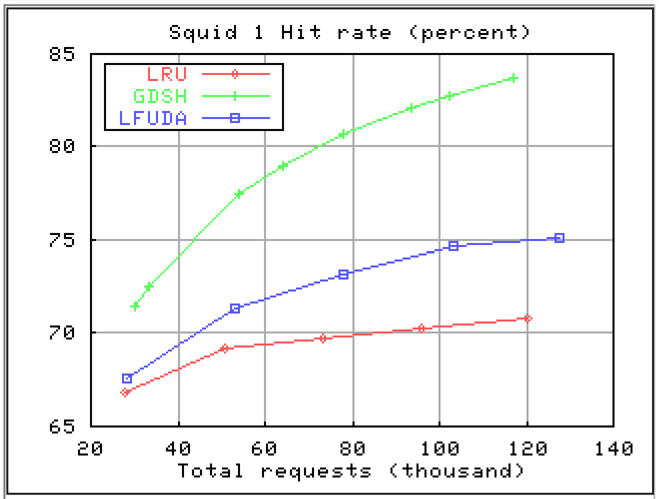


Figure 3: Squid 1 Hit Rate

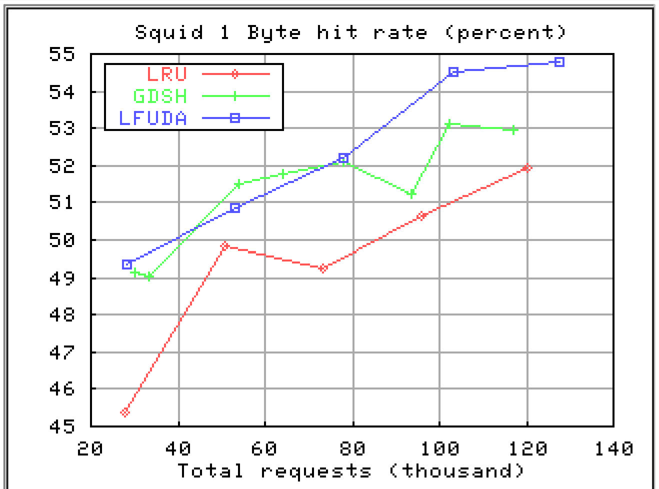


Figure 4: Squid 1 Byte Hit Rate

Note: if browsing on the web you can click on any image to view a full-size version.

We first examined the cache hit rate of the various replacement policies by varying the run time of the SPECWeb driver. We used a constant load factor that stressed the cache without overloading it (24 target ops/sec in our hardware configuration). The results in Figure 3 indicate that the GDSF policy achieved around 10-20% higher hit rate than LRU, with greater improvement at longer run length. By its focus on keeping more small popular documents the GDSF policy stored three times as many objects in the cache as LRU did.

The increase in hit rate as run length increased is due to SPECWeb load-generator’s behavior. The ratio of unique objects to the total number of requests decreased as more SPECWeb clients made HTTP requests of the cache. As a result the maximum possible hit rate and byte hit rate improved.

The byte hit rate of the policies in Figure 4 show that the LFUDA and GDSF policies both outperform LRU. We note that byte hit rate is not as high as hit rate, nor is the improvement with the new policies as great. One reason for this is that the SPECWeb workload ranked the popularity of files based upon size. There were only a few big popular files, but many big unpopular files. Thus the increase in hit rate was achieved by keeping small popular files which did not contribute much to byte hit rate.

The GDSF finding contradicted our findings in the simulator, which indicated that under a live workload the LRU policy should outperform GDSF in byte hit rate. The reason for the disparity is in the nature of the SPECWeb workload -- there are many more small popular files than we observe in real workloads. Said another way, real workloads exhibit a very heavy tail in the document size distribution, which leads to a few very large documents consuming a significant amount of the byte traffic. By caching these, the LRU policy outperformed GDSF in the simulator; however LFUDA outperformed LRU in byte hit rate in all cases.

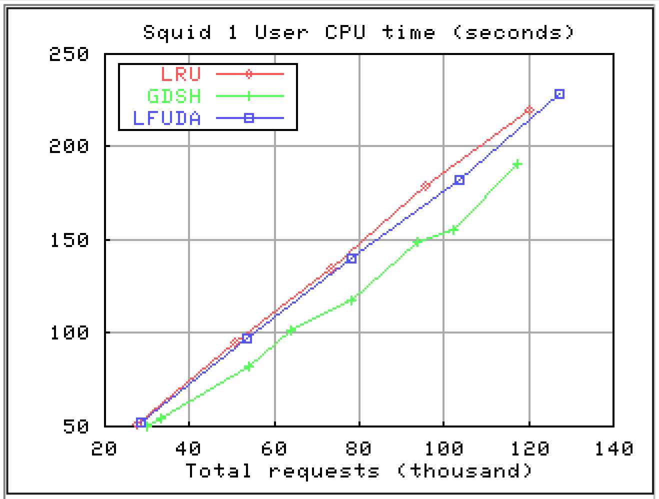


Figure 5: Squid 1 User CPU Utilization

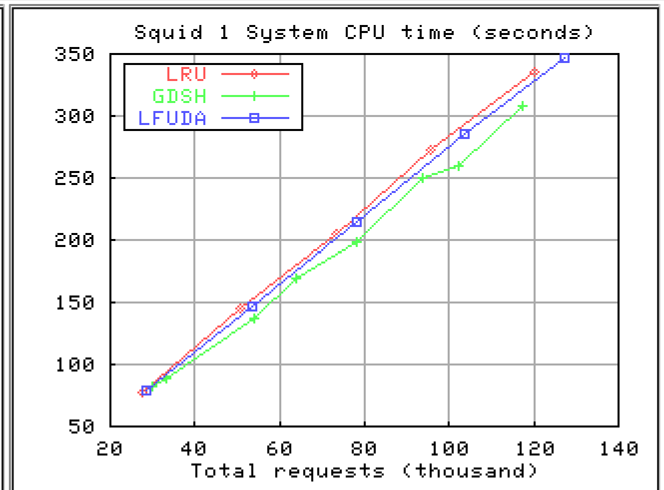


Figure 6: Squid 1 System CPU Utilization

We examined the user CPU usage (left) and system CPU usage (right) of the Squid processes while they were serving the requests reported in the previous figures. We observe that the new replacement policies actually reduce the user CPU demand, even though the measured version uses floating point division to compute a more complex function for replacement decisions. The reason for the result is evident when the user CPU demand is examined. By keeping the most popular documents in cache, a proxy does not have to do as much work replacing documents and later retrieving fresh copies of those documents over the network. Originally we had speculated that the savings also came from the reduction in disk I/O operations. Upon further analysis we discovered little difference in disk I/O or total bytes swapped in and out of memory. We conclude that I/O activity does not account for a significant CPU cost savings.

3.2 Squid 2 Validation

After implementing the replacement policies for Squid 2 we re-ran the SPECWeb benchmark to assess how they compared with the new list-based LRU implementation in Squid2. We expected that the hit rate and byte hit rate measurements would yield similar results since the replacement policy should control this factor more than other implementation details. An important question for this implementation was whether the CPU utilization with the heap implementation would be significantly different from the Squid 2 linked list-based replacement policy implementation.

Some of our initial observations from Squid 2 were difficult to explain. This led us to implement a heap-based LRU replacement policy in addition to the list-based LRU to better understand the effects of the changes we had made to the Squid 2 code (in particular to `storeMaintainSwapSpace` and its children). Upon further analysis we were able to understand why the policies behaved as they do. Some of the important differences between the standard and HPL replacement policies (GDSF, LFUDA, and LRU_HEAP) are:

- The HPL policies do not touch object metadata as often as the standard policy. They let memory usage grow to over the median water mark (between the high and low water marks) before making any replacement decisions. As a result they tend to use less CPU time than the standard LRU policy, which examines many more objects per object released.
- The HPL policies are more aggressive when evicting objects from the cache. When the memory usage gets high enough they will usually evict objects until the low water mark is reached. As a result they tend to run with fewer objects in cache as compared to the standard LRU policy, which uses the LRU reference age to maintain storage space closer to the high water mark.

In the following graphs, LRU_L indicates the original LRU linked-list policy, and LRU_H indicates the heap-based LRU reference implementation.

3.2.1 Hit Rates

Figure 7 presents the object hit rate of the replacement policies running under the same workload as in the Squid 1 characterization. In Squid 2 the GDSF policy shows improvement over LRU, as predicted by our simulation and the empirical validation with the SPECWeb workload: by keeping more smaller, more popular documents in cache the hit rate is maximized.

With LFUDA, keeping more popular documents also translates into an improvement in hit rate over LRU_H, however note that LRU_L was slightly better in hit rate than LFUDA. In our simulation, LFUDA achieved a higher hit rate than LRU, and in fact it achieves a noticeable higher hit rate than the equivalent LRU_H policy. But by utilizing what amounts to a larger memory area LRU_L is able to outperform LFUDA in this test.

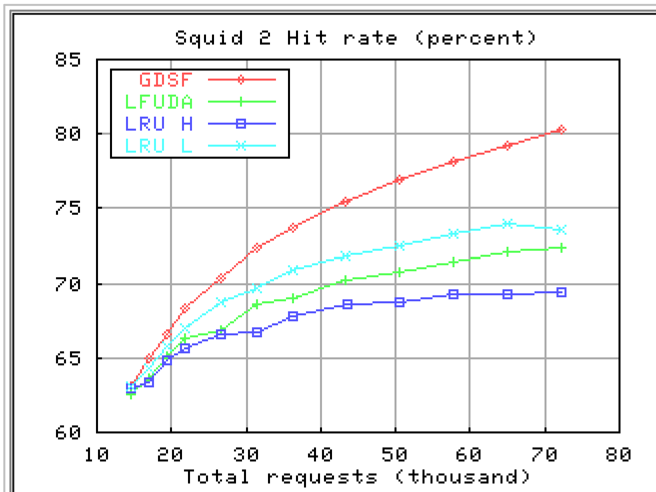


Figure 7: Squid 2 Hit Rate

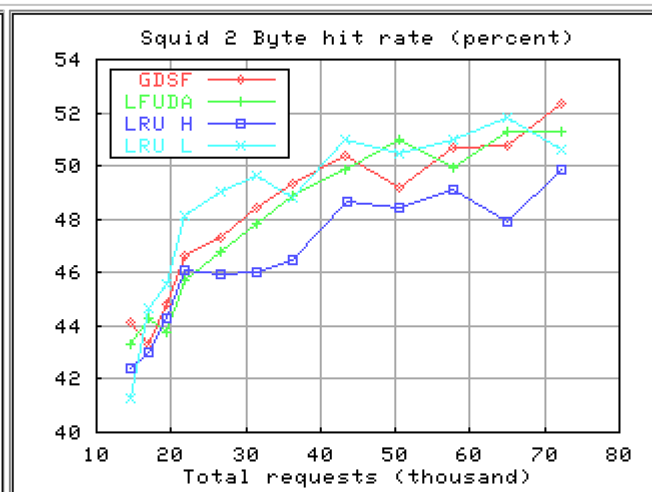


Figure 8: Squid 2 Byte Hit Rate

Figure 8 presents the byte hit rate of the four policies. From this graph the only conclusion we could draw was that all of the policies achieved roughly equal byte hit rate. Given the high variance in byte hit rate it is difficult to draw more substantial conclusions about which policy is optimal. From our simulation with document sizes drawn from an actual web workload the LFUDA policy achieves better byte hit rate than the other two policies, followed by LRU and finally GDSF.

The reason the variance is so large is that there are few large popular objects in the SPECWeb workload, and the workload for each run makes different requests of the cache. If a cache hit happens to occur on one of the large objects it has a noticeable effect on the byte hit rate for that run. In order to better understand the behavior of the replacement policies under a consistent workload we have run a separate characterization using an internal tool called `http` [HTTP], which is able to replay a request trace. The results from this experiment are presented in the Section 4.

3.2.3 CPU Utilization

Figures 9 and 10 present the user and system CPU utilization of the replacement policies. The results indicate that the heap-based implementation of the GDSF and LFUDA policies consume less user and system CPU time than the original LRU replacement policy in Squid 2. Interestingly the heap-based LRU implementation also consumes less CPU time than the original list-based LRU implementation. This is due to the optimizations made in `storeMaintainSwapSpace`, which performs fewer computations per released object and releases more objects on average per active invocation. (An "active invocation" is one in which work is done. Most invocations of this function only check the swap utilization and return without iterating through any metadata.)

The system CPU time is very nearly the same and less than half the user CPU time: Squid spends most of its time in user code, and of that time not much of it is directly attributable to the computational complexity of the replacement policy. The heap-based O(log N) LRU replacement policy consumes fewer CPU resources than the list-based O(1) LRU policy thanks to the optimizations described earlier.

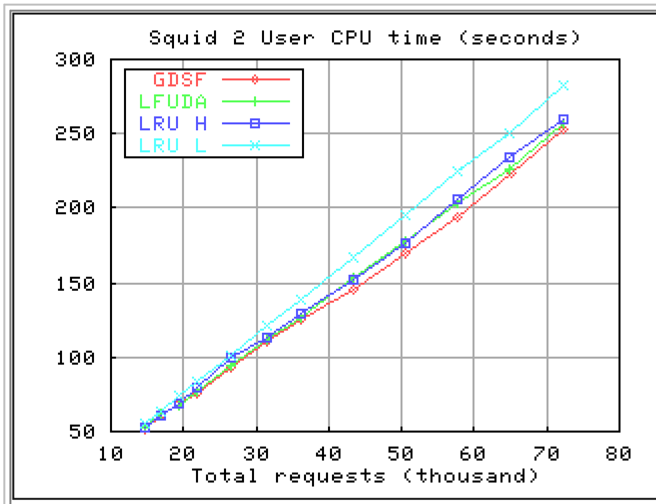


Figure 9: Squid 2 User CPU

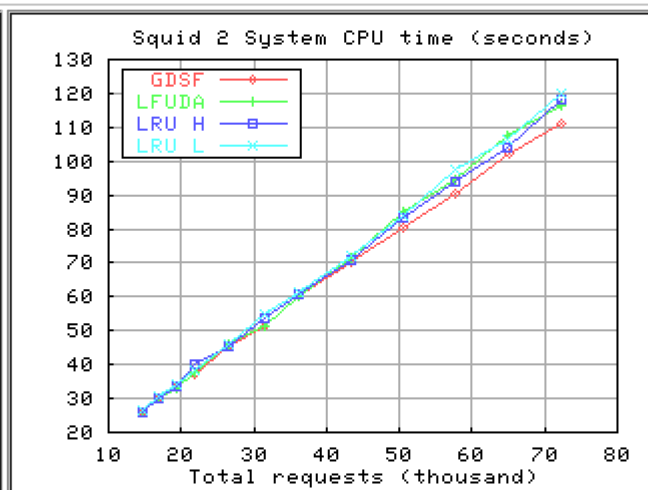


Figure 10: Squid 2 System CPU

3.2.4 Response Time

When we examine response time we see further evidence of the inefficiency in the original Squid 2 replacement policy. By comparing the two LRU policies we see that the list-based policy has higher and more variable response times. This is due to the extra time spent computing the LRU reference age and examining the same object multiple times without replacing or moving it.

Note that this extra time apparently seems to translate into a higher hit rate for LRU_L. However, as noted earlier the list-based policy also allows the cache to operate closer to the high water mark, so LRU_L has a larger effective cache space to work with than LRU_H does.

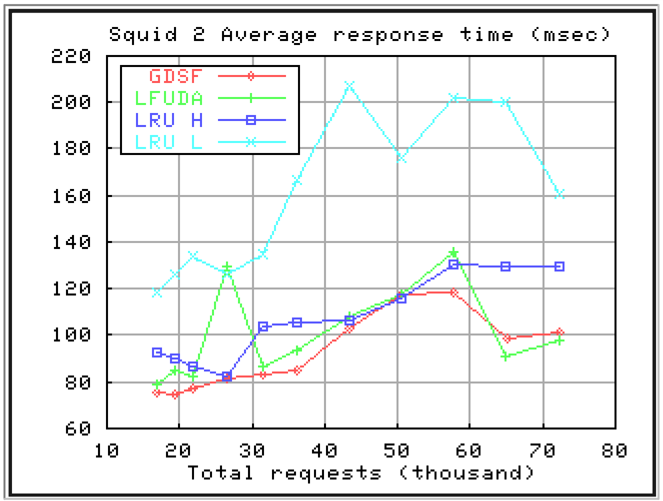


Figure 11: Squid 2 Response Time

4. Revalidation using `http`

In our SPECWeb validation we observed significant variation in the byte hit rate among the policies. There was also significant variation between successive runs of the same policy. We wanted to eliminate this variability by using a repeatable but SPECWeb-like workload. This section presents the results of that experiment.

The SPECWeb workload was statistically equivalent in terms of the request size and popularity distribution since we used the same workload configuration for all the tests. However, the requests made to the proxy varied pseudo-randomly from run to run, which can have a significant effect on the byte hit rate. When large, unpopular objects are requested they can replace a relatively large number of smaller objects. If such an object is re-referenced it will have a significant positive impact on the byte hit rate for that cache run since there are relatively few large objects but they consume a significant portion of the disk space and transfer bandwidth. The table below summarizes the SPECWeb file size and popularity distributions by class.

Size class	Object request rate	Byte request rate	Avg bytes transferred
0	35 %	1.82 %	787
1	50 %	17.5 %	5,315
2	14 %	46.5 %	50,290
3	1 %	34.1 %	516,400

Table 1: SPECWeb Request Distribution

From the table it is apparent why large objects have a pronounced impact on the byte hit rate. With this revalidation we wanted to eliminate the variance in the workload in order to create an apples-to-apples comparison of the policies. Section 4.1describes our approach to the problem. Section 4.2 describes the results of the revalidation. Section 4.3 summarizes our findings.

4.1 New Approach

In order to examine the behavior of the policies under a consistent workload we used the `http` tool [HTTP] to replay a log file (request trace) against each Squid 2 cache implementation. This tool is similar to the `httperf` workload generator [httperf], which is freely available as open source. However, it is not able to easily replay log files, so we used the original `http` tool instead.

The request log file we used was generated by an earlier SPECWeb test on a Squid cache that made approximately 100,000 requests. We partitioned the trace into sections of 5,000 and 10,000 requests and replayed those traces against each of the four cache replacement policy implementations. The command we used to replay the log was as follows:

```
http -log $log -max ‘wc -l < $log‘ -reqs 10 -quiet
```

This command replays a log trace, `$log`, making each requests just once (`-max ‘wc -l < $log‘`), and making up to 10 requests in parallel (`-reqs 10`) the `-quiet` option causes `http` not to log every request it makes to the origin server. The `http` tool is able to replay any trace that has the pattern " `GET http://uri`" in it, which the Squid `access.log` file from each of our previous SPECWeb runs has. We selected several logs from previous SPECWeb runs, processed them to contain only the URLs, split the URL file into trace segments, and then replayed each of them. The trace segments had the following properties.

- The first trace contained 10,000 requests. This was to warm up the cache, since the policies all behave about the same working against a cold cache: they are storing all objects. Only after about 10,000 requests are any replacement decisions being made.
- The next four traces contained 5,000 requests. This was to see detail on the low end of the graphs.
- All but the final trace contained 10,000 requests.
- The final trace contained however many requests were left over (under 10,000).

4.2 Results

The figures below indicate the hit rate and byte hit rate for the four policies considered. These are the same four policies as described earlier; the tests used the same Squid executables used in those tests.

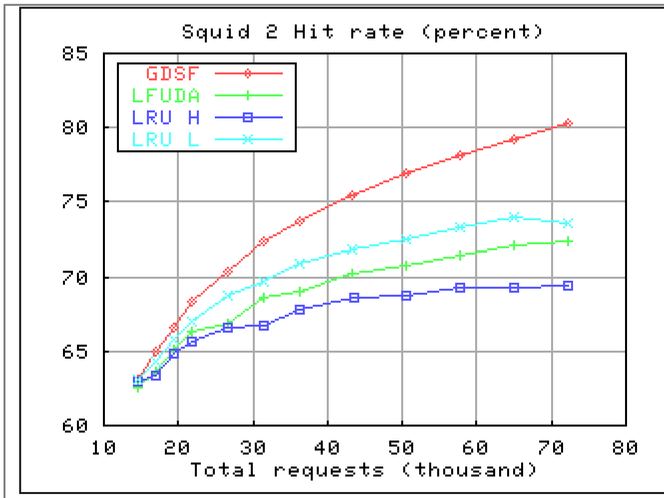


Figure 1: SPECWeb Hit Rate

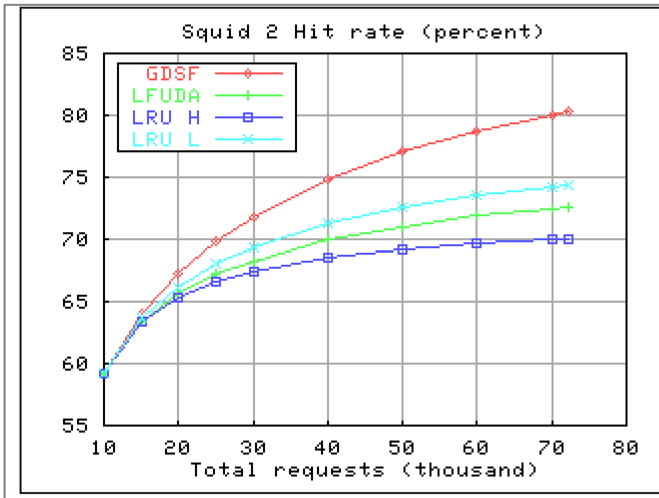


Figure 2: `http` Hit Rate

Figures 1 and 2 show the hit rate of the Squid 2 implementations under the SPECWeb workload and the `http` workload. In hit rate there is little difference between the original SPECWeb validation and this revalidation: the relative ordering has not changed and the shape of the curves is broadly similar. There is greater variance in the policies in the SPECWeb version and note that the run length of the `http` test is slightly greater.

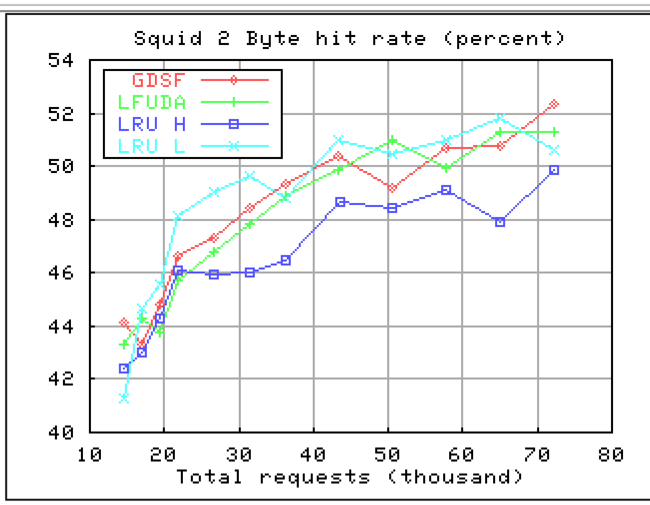


Figure 3: SPECWeb Byte Hit Rate

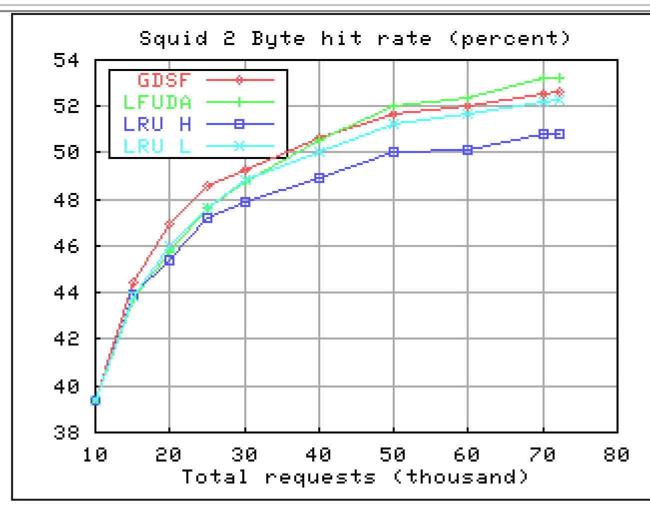


Figure 4: `http` Byte Hit Rate

Figures 3 and 4 present the byte hit rate under SPECWeb and `http` workloads. Here we see the behavior of each of the policies appears more stable under an identical workload. (The workload for the sweep in Figure 4 is the same as the GDSF sweep in Figure 3, using the log generated during that run.) In these figures we can see that the LFUDA policy has the best byte hit rate of all policies given the same workload, although GDSF is better during the early part of the trial (before a sufficient number of large, popular documents have been accessed).

We also examined the response time of the various policies under the `http` workload. (Given our test setup we were not able to extract CPU time measurements since we did not shut down the Squid server between trace segments.) In Figures 5 and 6 we see again that the consistent workload produces much smoother results. With this run we are able to discern between the GDSF and LFUDA policies to determine which has the best response time, and note that the original LRU replacement policy implementation has efficiency problems, although not to the extent we saw with the original SPECWeb test.

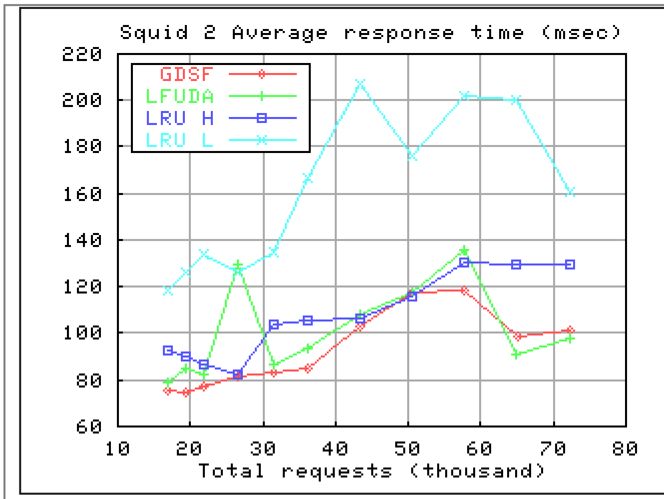


Figure 5: SPECWeb Response Time

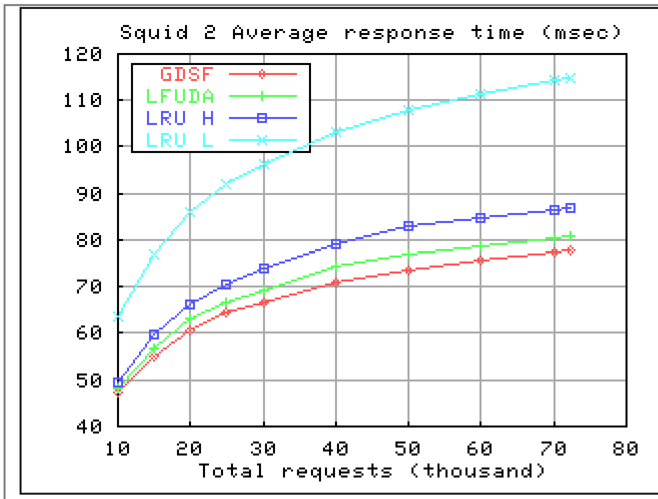


Figure 6: `http` Response Time

4.3 Summary of Findings

These results broadly corroborate our SPECWeb experiments and further refine them by requesting the same set of URLs in the same order from each of the replacement policy implementations. With the consistent workload we see that the GDSF policy is a consistently significant improvement over LRU in Squid, as was predicted through our earlier trace-driven simulation.

We have run this validation using several of the traces from our initial SPECWeb benchmark tests all with similar results. There are variations between the runs, but with a single data set the curves we generated are substantially the same as the ones presented here: the curves have the same shape and the policies all have the same relative order.

5. Summary and Conclusions

The choice of a cache replacement policy can have a marked effect on the network bandwidth demand and object hit rate in a cache. From our work we have observed that the tremendous breadth in a real web working set makes achieving a high hit rate difficult, and especially a high byte hit rate. However there is still room for improvement. We implemented two frequency-based cache replacement policies in Squid and observed their effects under a synthetic workload generated by SPECWeb. The empirical results corroborate our earlier simulation results. We then observed their effects under a fixed workload and were able to better discern performance characteristics among the policies. In hindsight, SPECWeb was not the best workload driver, other than to generate a working set.

These results are encouraging in that a more sophisticated (for example heap-based) replacement policy can be implemented in a real cache implementation without degrading the cache’s performance. This follows intuition, since the dominant component of the proxy cache workload is I/O (network and disk). Since the CPU is not the bottleneck resource, it is worthwhile to invest a few extra cycles in the replacement policy to eliminate disk or network I/O. The Squid experience reported here is likely true as well for other cache implementations; we have anecdotal evidence that other cache servers have successfully implemented more sophisticated cache replacement algorithms.

We would like to further explore the performance of the new cache replacement policies under a more realistic proxy workload. The best way to do this is by using a benchmark able to generate a workload appropriate for a proxy, such as the Wisconsin Proxy Benchmark [AC98] or the Polygraph Benchmark [Poly]. We also intend to run the modified cache on our internal proxy cache and observe its operation over a longer duration than the one-hour maximum SPECWeb runs, and with a live workload.

We continue to look for more optimal replacement policies, particularly to save byte hit rate. A first step is to determine the theoretical maximum hit rate and byte hit rate for caches of various sizes to determine how close each cache replacement policy is to the maximum -- and therefore how much room for improvement exists. Unfortunately, this problem reduces to the bin packing problem and is therefore NP complete! (The proof of which this margin is too small to contain.) Nevertheless, we may be able to identify appropriate heuristics to approximate the optimum cache packing in our search for better replacement policies. One unfortunate limiting factor is the breadth and turnover of the working set, which in any case will bound the benefit of caching for bandwidth reduction.

It has been noted that the cache replacement policy simply does not matter; that by adding more disk space you make up for a few percent improvement that can be gained in improving the replacement policy. While adding resources is always an attractive option, we believe it is nevertheless worthwhile to make the best use of the resources at hand, however they may grow.

We intend to submit our cache replacement policy implementation to the Squid open source community and to freely license them to other cache vendors.

As bandwidth becomes cheaper and more available caching will play a greater role in reducing access latency and origin server demand. To achieve the greatest latency reduction and origin server demand a cache should strive to optimize cache hit rate; but while (or where) bandwidth is dear the cache must focus on improving byte hit rate. For this reason a configurable cache replacement policy is advantageous to cache administrators.

Furthermore, the latency implication of cached object consistency checking become more significant as local bandwidth and access latency improve. Wide area bandwidth is improving as well, but wide area latency is bounded by the speed of light and the number of packet round trips that must be made to satisfy a user request. A consistency check for a small object takes about as long as retrieving the object in the first place. We intend to further analyze this, and are exploring techniques for improving object consistency in large scale wide area distributed systems.

6. Acknowledgments

Godfrey Tan contributed to the Squid 1 work reported in this paper during his 1998 summer internship at HP Labs. His efforts were instrumental in modifying the SPECweb98 tool, and validating the Squid 1 implementation of the cache replacement policies. We also appreciate the comments of the anonymous reviewers, which have improved the quality of this paper, and the dedication of the Squid community to making the Squid proxy cache implementation widely available and well supported.

7. References

AC98
: J. Almeida and P. Cao, "Measuring Proxy Performance with the Wisconsin Proxy Benchmark", Technical Report, University of Wisconsin-Madison, April 1998.

ACDFJ99
: M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin, "Evaluating Content Management Techniques for Web Proxy Caches", in Proceedings of the 2nd Workshop on Internet Server Performance (WISP ’99), Atlanta GA, May 1999

AFJ99
: M. Arlitt, R. Friedrich, and T. Jin, "Workload Characterization of a Web Proxy in a Cable Modem Environment ", to appear in Performance Evaluation Review, August 1999.

C98
: L. Cherkasova, "Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching Policy", Technical Report HPL-98-69R1, Hewlett-Packard Laboratories, Nov. 1998.

CI97
: P. Cao and S. Irani, "Cost-Aware WWW Proxy Caching Algorithms", USENIX Symposium on Internet Technologies and Systems, Monterey, CA, pp. 193-206, Dec. 1997.

DFK97
: F. Douglis, A. Feldmann, and B. Krishnamurthy, "Rate of Change and Other Metrics: A Live Study of the World-Wide Web", USENIX Symposium on Internet Technologies and Systems, Monterey, CA, pp 147-158, Dec. 1997.

HTTP
: T. Jin, "HP-UX 10.x Manual for http(1)", (HP internal documentation only) http://granite.hpl.hp.com/cgi-bin/parse/man2html?http.1

httperf
: D. Mosberger, T. Jin, "httperf -- A Tool for Measuring Web Server Performance", http://www.hpl.hp.com/personal/David_Mosberger/httperf.html

Poly
: Web Polygraph proxy performance benchmark, http://www.ircache.net/Polygraph/

SPECWeb
: The Workload of SPECWeb Benchmark, http://www.spec.org/osg/web96/workload.html

Squid
: Squid Internet Object Cache, http://squid.nlanr.net/

TMW97
: K. Thompson, G. Miller, and R. Wilder, "Wide-Area Internet Patterns and Characteristics", IEEE Network, Vol. 11, No. 6, November/December 1997.

WCW98
: 3rd International WWW Caching Workshop, http://wwwcache.ja.net/events/workshop/

WCW99
: 4th International WWW Caching Workshop, http://www.ircache.net/Cache/Workshop99/