Improving Proxy Cache Performance-Analyzing Three Cache Replacement Policies

John Dilley, Martin Arlitt
Internet Systems and Applications Laboratory
HP Laboratories Palo Alto
HPL-1999-142
October, 1999

World-Wide Web, proxy caching, replacement policies, performance evaluation

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 determines which objects remain in cache and which are evicted to make room 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.

Internal Accession Date Only

Improving Proxy Cache Performance - Analyzing Three Cache Replacement Policies

John Dilley, Martin Arlitt - HP Laboratories, Palo Alto, CA, USA

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 determines which objects remain in cache and which are evicted to make room 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.

1.0 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 throughout the Internet, from the user’s browser cache through local proxy caches and backbone caches, to the so called reverse proxy cache located near the origin of web content. These caches different sets of users and have slightly different goals.

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 (the standard HTTP port) 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 must determine whether each object it serves is *fresh* or not; if not, the cache validates the object with its origin server, otherwise it serves it directly. This freshness decision is typically based upon the object’s last modification time and the time of last retrieval or validation. The validation returns a fresh copy of the object or a status code indicating the object has not changed.

1.1 The Role of a 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 a set of objects to evict in order to make room for newly requested 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 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 behavior, or the 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. We 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 we implemented and validated. The remainder of this article summarizes the implementation and validation of these policies in the Squid open source proxy cache.

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 cache studies in computer system architecture 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; conversely 70% of all bytes returned to users were retrieved across the external network.

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 trade off against each other. In order to maximize object hit rate it is better 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 large popular objects. It is clearly preferable to keep objects that will be popular in the future and evict unpopular ones; the trade-off 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 also can be accomplished in constant time by moving the object to the head of a doubly-linked list.

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 with 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 formerly popular object becomes unpopular, it will remain in the cache a long time, preventing other newly (or slightly less) 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 two new policies which are examined in the next section.

2.0 New Replacement Policies

The most widely deployed proxy cache server today seems to be the *Squid* open source software package [Squid]. Squid is highly portable, freely available in source code form, and has an active, experienced developer community that provides free support. The availability of source code and technical assistance led to our choice of Squid to experiment with our new cache replacement policies.

The replacement policy that has been used by Squid is the LRU policy. We designed and implemented in Squid two new variants of the LFU and GDS policies, based upon analysis and trace-based simulation of the web workload mentioned earlier [ACDFJ99].

- *GDS-Frequency* (GDSF): This variant of the Greedy Dual-Size policy takes into account frequency of reference in addition to size. This policy is optimized to keep more popular, smaller objects in cache 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 below for LFUDA.
- *LFU with Dynamic Aging* (LFUDA): This variant of LFU uses dynamic aging to accommodate shifts in the set of popular objects. In the dynamic aging policy, a cache age factor is added to the reference count when an object is added to the cache or an existing object is re-referenced. This prevents previously popular documents from polluting the cache. Instead of adjusting all key values in the cache, as some aging mechanisms require, the dynamic aging policy simply 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 always less than or equal to the minimum key value in the cache. This also prevents the need for parameterization of the policy, which LFU-Aging requires.

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 2

We implemented our cache replacement policies in two separate versions of Squid, first version 1.1.20, then 2.2.STABLE3. These versions had significantly different implementations of the replacement algorithm. The earlier version was much easier to implement our policies, however the later version is what is now being used in the field. We implemented and validated our policies in Squid 2 and have since released the source modifications back to the Squid community. Our code modifications will be available in the 2.3 version of Squid. Details of our implementation experiences can be found in [DAP99].

3.0 Replacement Policy Validation

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 the two parameterless replacement policies described earlier. We implemented the policies in a cache 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 then 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.

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. 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 lev-

els. 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 system and pointed it at the Squid HTTP accelerator port on the system under test. We assessed 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. We did not use a cache warm-up period in our tests.

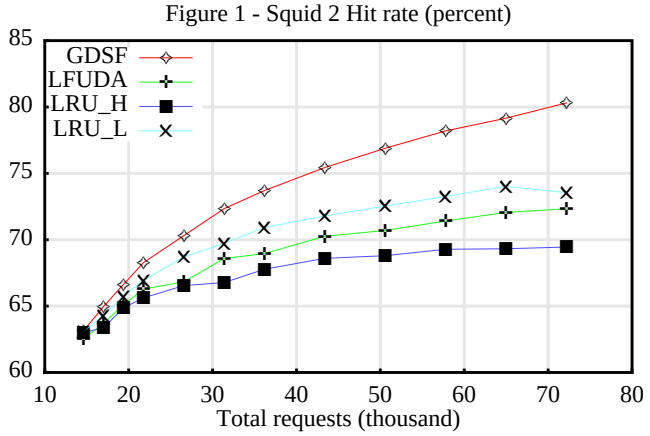
3.1 Squid 2 Validation

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 cache maintenance routine when replacing the list-based LRU policy with our new heap-based policies. Some of the important differences between the list-based (LRU) and heap-based replacement policies (GDSF, LFUDA, and LRU_HEAP) are:

- The new policies do not touch object metadata as often as the standard policy. They let memory usage grow to a high water mark 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 new policies are more aggressive when evicting objects from the cache. When the memory usage is high enough they will evict objects regularly until the low water mark is reached. As a result these policies tend to run with fewer objects in cache as compared to the standard LRU policy, which keeps cache storage space closer to the high water mark.

In the following graphs, LRU_L indicates the original LRU linked-list policy; LRU_H indicates the heap-based LRU reference implementation; and GDSF and LFUDA are the new policies described above.

Figure 1 - Squid 2 Hit rate (percent)



3.1.1 Hit Rates

Figure 1 presents the object hit rate of the replacement policies running under the SPECweb workload. The GDSF policy shows improvement over LRU, as predicted by our simulation: by keeping more smaller, more popular documents in cache the hit rate is improved.

With LFUDA, keeping more popular documents also improves in hit rate over LRU_H, however note that LRU_L has a slightly better 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 LRU_H policy. But by utilizing what amounts to a larger memory area LRU_L is able to outperform LFUDA in this test.

Figure 2 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

Figure 2 - Squid 2 Byte hit rate (percent)

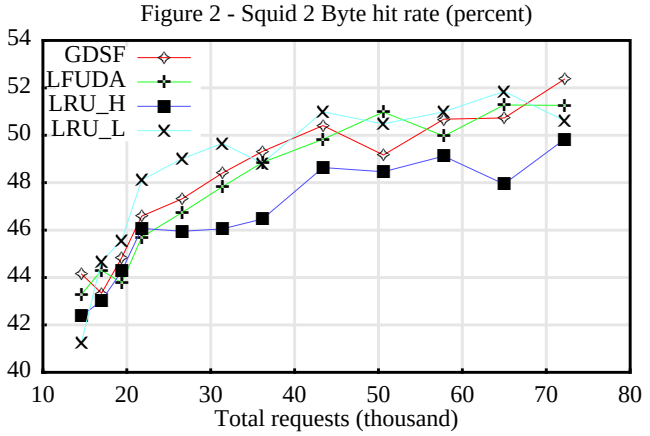
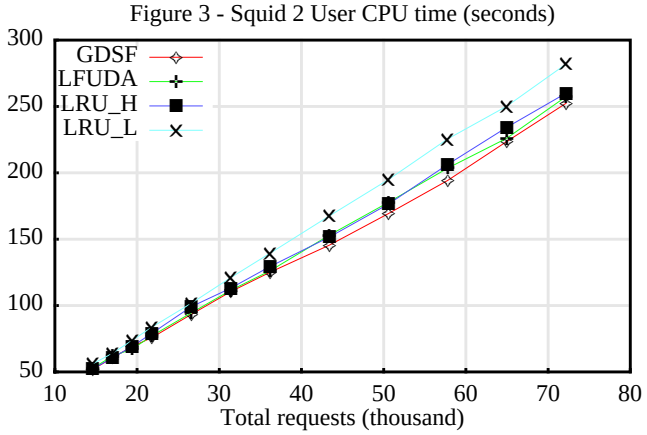


Figure 3 - Squid 2 User CPU time (seconds)



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 a tool that is able to replay a request trace.

3.1.2 CPU Utilization

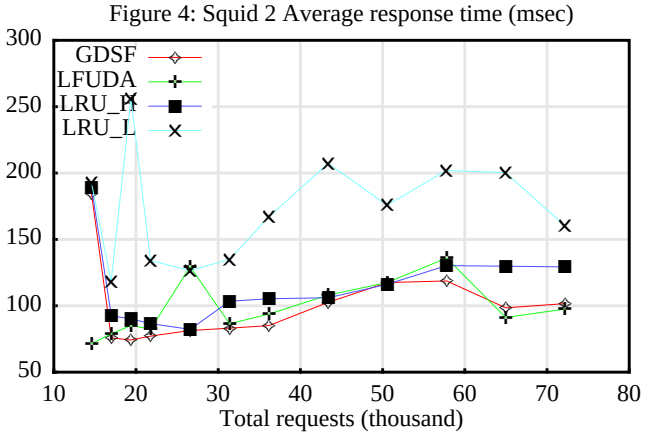
Figure 3 presents the user CPU utilization of the replacement policies. The results indicate that the heap-based implementation of the GDSF and LFUDA policies consume less user and system CPU time than the original LRU replacement policy in Squid 2. Interestingly the heap-based LRU implementation also consumes less CPU time than the original list-based LRU implementation. This is due to the optimizations made in running the replacement algorithm, 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 check the storage utilization and return without examining any objects.)

System CPU time is very nearly the same for all implementations and less than half the user CPU time. We conclude that Squid spends most of its time in user code, and of that time not much of it is attributable to the computational complexity of the replacement policy.

3.1.3 Response Time

When we examine response time we see further evidence of inefficiency in the original Squid 2 replacement policy. By comparing the two LRU policies we see that the list-based

Figure 4: Squid 2 Average response time (msec)



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 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. This can be altered by setting the low water mark higher for the heap implementation, but we did not do that because we wanted the cache parameters to be identical for all of our tests.

4.0 Revalidation Using Log Replay

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 still SPECweb-like workload. This section presents the results of that experiment.

The offered workload in each of the SPECweb runs 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.

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.

Table 1 SPECweb Request Distribution

Class	% Reqs	% Bytes	Avg Bytes
0	35 %	1.8 %	787
1	50 %	17.5 %	5,315
2	14 %	46.5 %	50,290
3	1 %	34.1 %	516,400

4.1 New Approach

In order to examine the behavior of the policies under a consistent workload we used an internal tool called http to replay a log file (request trace) against each Squid 2 cache implementation. 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.

4.2 Summary of Findings

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

We ran 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 are substantially the same as the ones presented here: they have similar shape (but smoother) and the policies have the same relative order.

5.0 Conclusions and Future Work

The choice of a cache replacement policy can have a marked effect on the network bandwidth demand and object hit rate of a proxy cache. From our earlier work we observed that the tremendous breadth of a real web workload makes achieving a high hit rate difficult, and especially a high byte hit rate. However there is room for improvement. We implemented two frequency-based cache replacement policies in the Squid open source proxy cache 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.

As bandwidth becomes cheaper and more available caching will play a greater role in reducing access latency and origin server demand than simply bandwidth. 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. Our implementation as released in Squid 2.3 allows this choice through the standard cache configuration mechanism.

Furthermore, the impact of consistency validation of cached objects become more significant as local bandwidth and access latency improves. Wide area bandwidth is improving as well, but wide area latency is bounded by the speed of light and the number of network round trips that must be made to satisfy a user request. A consistency check for a small (1KB) object usually takes 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.

6.0 Acknowledgments

Godfrey Tan contributed to the implementation of these policies in Squid 1. His efforts were also instrumental in modifying the SPECweb98 tool and validating the Squid 1 implementation of these policies. Stéphane Perret also contributed to the development and documentation of this work.

We also appreciate the comments we received from Terence Kelley and Yee Man Chan, and the dedication of the Squid community to making the Squid proxy cache implementation widely available and well supported.

7.0 REFERENCES

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”, in Performance Evaluation Review, August 1999.

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

DAP99. J. Dilley, M. Arlitt, S. Perret, “Enhancement and Validation of Squid’s Cache Replacement Policy”, Technical Report HPL-1999-69, Hewlett-Packard Laboratories, May 1999.

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, December 1997.

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

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