Improvement of the Neural Network Proxy Cache Replacement Strategy

Hala ElAarag and Sam Romano
Department of Mathematics and Computer Science
Stetson University
421 N Woodland Blvd, Deland, FL 32723
{helaarag, sroman}@stetson.edu

ABSTRACT

As the Internet has become a more central aspect for information technology, so have concerns with supplying enough bandwidth and serving web requests to end-users in an appropriate time frame. Web caching help decrease bandwidth, lessen user perceived lag, and reduce loads on origin servers by storing copies of web objects on servers closer to end users as opposed to forwarding all requests to the origin servers. Since web caches have limited space, web caches must effectively decide which objects are worth caching or replacing for other objects. This problem is known as cache replacement. We used neural networks to solve this problem and proposed the Neural Network Proxy Cache Replacement (NNPCR) method. In this paper we propose an improved strategy of NNPCR referred to as NNPCR-2. We implemented NNPCR-2 in Squid proxy server and compared it with four other cache replacement strategies. In this paper we use 84 times more data than NNPCR was tested against and present exhaustive test results for NNPCR-2 with different trace files and neural network structures. Our results demonstrate that NNPCR-2 made important, balanced decisions in relation to the hit rate and byte hit rate; the two performance metrics most commonly used to measure the performance of web proxy caches.

Keywords: web caching, proxy server, Squid, Neural network, NNPCR, cache replacement strategies

1. Introduction

Due to the large increase in Internet traffic, proxy servers are designed with three goals: decrease network bandwidth, reduce client perceived lag, and reduce loads on the origin servers [1, 2]. Several problems emerge from the fact that there is a limited amount of hard disk space that proxies can save objects onto. Our main focus is the cache replacement problem. The most commonly known cache replacement strategies are Least Frequently Used (LFU) and Least Recently Used (LRU). Podlipnig et al. [1] has done well to not only list well-known cache replacement strategies, but also categorize the strategies into five groups: Frequency Based, Recency Based, Frequency/Recency Based, Function-Based, and Randomized.

As we will demonstrate later in this paper, many strategies suffer from not being able to adapt dynamically to ever changing levels of request stream characteristics such as temporal and spatial locality. Temporal locality is the measure of how likely an object is to appear again in a request stream after being requested within a time span, while spatial locality is the likelihood that an object will appear again based on how often it’s been seen before [2]. In [6] we demonstrated that many strategies fail to separate characteristics of the request stream from characteristics of individual web requests and as such also miss one, two, or all three goals of a web proxy server in comparison to other strategies.

Neural networks were applied to the web proxy cache replacement problem in 2006 as a novel cache replacement strategy by Jake Cobb and Hala ElAarag [3, 4]. The resulting framework was named the Neural Network Proxy Cache Replacement (NNPCR) method. While consistently outperforming LRU and LFU, it was unknown how well NNPCR would fair against more advanced, equivalent strategies in a practical environment. In this research, we implemented NNPCR in the Squid Proxy Cache version 3.1 as well as the function-based strategy, M-Metric [10] to see how they would fair against the existing strategies implemented in Squid.

The rest of this paper is structured as follows. In section 2, we describe current cache replacement strategies implemented in Squid, along with M-Metric and NNPCR. Section 3 discusses the many improvements we propose for the implementation of NNPCR, dubbed NNPCR-2 and its implementation in Squid version 3.1. Section 4 describes the testing setup for Squid and how each strategy was benchmarked. Section 5 presents the results and observations of our tests of various cache replacement strategies covered in this paper including NNPCR-2. We conclude our results in section 6.

2. Related Replacement Strategies

Squid [5] is a good test bed for NNPCR. As the need for web proxy caching increases, Squid grows more popular in use and in support, and the added support and popularity has lead to a rather stable product.
2.1 Squid’s Implemented Cache Replacement Strategies

Squid comes with three strategies already implemented: LRU, LFU Dynamic Aging (LFU-DA), and Greedy Dual Size-Frequency (GDSF). In the rest of this section, we will provide a summary of each of these three strategies. Table 1 contains commonly used variables and their descriptions, which will be referred to throughout the rest of this paper. If a strategy uses a variable not defined in the table, then it will be defined in their summary. Any use of the logarithmic function, symbolized as \( \log \), is assumed to be of base 2, unless otherwise specified.

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S_i )</td>
<td>Size of an object ( i )</td>
</tr>
<tr>
<td>( T_i )</td>
<td>Time object ( i ) was last requested</td>
</tr>
<tr>
<td>( \Delta T_i )</td>
<td>Time since object ( i ) was last request</td>
</tr>
<tr>
<td>( F_i )</td>
<td>Frequency counter of object ( i )</td>
</tr>
<tr>
<td>( \Delta F_i )</td>
<td>Number of references to occur since last time object ( i ) was referenced</td>
</tr>
<tr>
<td>( C_i )</td>
<td>Cost of object ( i )</td>
</tr>
<tr>
<td>( R_i )</td>
<td>Request value of object ( i )</td>
</tr>
</tbody>
</table>

- **LRU**: This algorithm removes the least recently used (referenced) object, or in other terms the object with the largest \( \Delta T_i \) or smallest \( T_i \). A simple linked list allows this algorithm to be efficiently implemented and is referred to as a LRU list.

- **LFU-DA** [9]: Standing for Least Frequently Used – Dynamic Aging, LFU-DA attempts to remove cache pollution induced by LFU. A common problem that occurs is with hot objects. These objects become popular incredibly quick, and so their frequency counts skyrocket, and then are no longer requested. In a complete LFU strategy, these objects would expire long before the replacement method would remove them. As such, LFU-DA ages the frequency counts of older objects in order to account for this issue. Upon a request to object \( i \), its value, \( R_i \), is calculated as:  
  \[
  R_i = F_i + L \tag{1}
  \]
  where \( L \) is a dynamic aging factor. Initially \( L \) is set to 0, but upon the removal of an object \( i \), \( L \) is set to \( R_i \). This strategy removes the object with the smallest \( R_i \) value.

- **GDSF** [9]. An extension of GD-Size, this uses the frequency information and cost information as well as size to define a request value. The request value is defined as:  
  \[
  R_i = \frac{F_i C_i}{S_i} + L \tag{2}
  \]
  \( L \) is an aging factor used exactly like in LFU-DA. The advantage in this method is that objects which are beneficial to caching (typically with higher latencies) can be cached and decrease the network bandwidth. However, in Squid, it assumes what is called the uniform cost model, that is, for all objects, \( C_i = 1 \). As a result, the implementation of GDSF in Squid loses this advantage.

2.2 Implemented Cache Replacement Strategies

In this section we provide a description of the strategies we implemented within Squid version 3.1. We first present M-Metric [10], a function-based method, that was selected because of the interesting way it combines several characteristics together to make an assessment of the value of an object. The second strategy we implemented was NNPCR, as it was the focus of this research.

- **M-Metric** [10]: This strategy takes three parameters: \( f, s, \) and \( t \). With these in mind, it defines the request value as,  
  \[
  R_i = F_i f^* S_i^* \Delta T_i^* \tag{3}
  \]
  \( f \) should be positive as to give weight to popular objects. A positive \( s \) value will give higher weight to larger objects, while a negative value will give higher weight to smaller objects. \( t \) reflects how much recency is taken into account. A positive value gives weight to older objects, while a negative value will result in recent objects taking precedence over older ones. Based on the parameter values, this algorithm will decide exactly like LRU (\( f = 0, s = 0, t < 0 \)), LFU (\( f > 0, s = 0, t = 0 \)), and SIZE (\( f = 0, s < 0, t = 0 \)). One large difference between M-Metric and most other function based methods, however, is the lack of an aging factor. As a result, in an ideal cache setup, the value of an object in M-Metric should be updated each time a decision on the heap must be made (right before removing an object from the cache).

- **NNPCR** [3, 4]: Uses neural networks to rank objects in the range \([0.0, 1.0]\). Has a range of different inputs which it can be trained for. Cobb et al. [11] suggested several ranges for parameters to use when training the neural networks. Fixed parameters are presented in Table 2, while ranges for variable parameters are presented in Table 3. During training, a target value of 1.0 is assigned to a pattern if the particular web-object the request represents is requested within the training set afterwards, otherwise it is assigned a target value of 0.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Neural Network Model</td>
<td>Fully connected, feed-forward MLP model</td>
</tr>
<tr>
<td>Hidden Layers</td>
<td>2</td>
</tr>
<tr>
<td>Normalized Input Range</td>
<td>([-0.9, 0.9])</td>
</tr>
</tbody>
</table>
There are three inputs that NNPCR requires: Recency, Frequency and Size. Of the three, recency is the most troublesome of the parameters to normalize. Clearly, it’s possible that the highest recency value could be up to a day, week, even more. However, it’s unlikely that most objects will see a recency value that high, which means when we normalize our value, most objects will clutter around -0.9, which will not allow the neural network to be able to generalize very well. It may get to the point where the neural network would just concentrate on the other two inputs, namely, frequency and size respectively, and zero out the weights to the recency input node if it did not tell enough information. By zeroing out the weights the neural network would have nullified the purpose of using recency as an indicator for cacheability.

To counter this issue, we decided to implement a sliding window around the time a request is made. With this in mind, we could train our neural network around this sliding window, and redefine the semantic of our output value to go according to this sliding window. In NNPCR [11], we could say that the sliding window had a length of 2 hours, since all the testing and training was done within two hours. That means any output by the neural network was a judgment for within the next couple of hours.

To formally define what we mean by a sliding window, the system administrator would have to train and configure the neural network based on some sliding window length generally set in milliseconds for our training program and seconds for Squid version 3.1. The sliding window of a request is thus the time before and after when the request was made. Within this sliding window we can examine the frequency count during training, and the recency time. As a general rule of thumb, the sliding window should be around the mean time that an object generally stays in a cache. From here on, the Sliding Window Length will be referred to as SWL. Other symbols for object characteristics are shown in Table 1.

With the sliding window in mind, we update the recency value at each request to an object. When an object is requested, we set the recency value to:

\[
\text{recency}(x) = \begin{cases} 
\max(\text{SWL}, \Delta T) & \text{if object } i \text{ was requested before } \\
\text{SWL} & \text{otherwise}
\end{cases}
\]  

With this method, we then normalize the values into the range of [-0.9, 0.9] by a typical linear normalization formula such as shown in Equation 5.

\[
x_{\text{norm}} = \frac{x_{\text{actual}} - \min_{\text{data}}}{\max_{\text{data}} - \min_{\text{data}}} \cdot (\max_{\text{norm}} - \min_{\text{norm}}) + \min_{\text{norm}}
\]  

For the frequency input value it is relatively simple to decide on a threshold value. We must choose a frequency high enough that most objects won’t ever reach it, but low enough so that the input values are dispersed evenly across the range [-0.9, 0.9]. In [6], we discussed how methods such as the Cubic Selection Scheme (CSS) [12] and Pyramidal Selection Scheme (PSS) [13] have a maximum frequency

### Table 3: NNPCR Variable Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Suggested Value(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Learning Rate</td>
<td>0.005 – 0.05</td>
</tr>
<tr>
<td>Momentum Rate</td>
<td>0.2 – 0.6</td>
</tr>
<tr>
<td>Nodes Per Hidden Layer</td>
<td>3 – 10</td>
</tr>
<tr>
<td>Training Epochs Allowed</td>
<td>500 - 2000</td>
</tr>
</tbody>
</table>

### 3.1 NNPCR-2

While simulations demonstrated that NNPCR would achieve high performance rates, there were several limitations we encountered in this research. NNPCR had a 2-hour training set and then a 2-hour testing set. More favorably training and testing should be for longer periods. In this way, we can ensure that NNPCR-2 would perform well for a long period of uptime.

More than anything else, the recency input is heavily related to the running time of the strategy. This even affects the target output for training the strategy. For instance, if an object has a mean time between requests of 4 hours, is it worth it to cache the object? According to NNPCR, each of these requests when trained by the neural network would have a target value of 1.0 despite the fact that the object would most likely expire from the cache well before.

NNPCR was also able to read in the trace files prior to running each training/simulation session and found the maximum recency, frequency, and size values. This is clearly impossible in a running implementation. So clearly, we must define thresholds for each of the inputs and cap them if they happen to go above so that we can normalize our values into the input range [-0.9, 0.9] still. However, by adding more parameters/thresholds, we now make NNPCR more parameterized than before, taking away from its dynamic characteristics. In the following sections, we will describe the new method of normalizing our inputs, as well as redefining our target value.

### 3.1.1 Input Normalization Methods and the Sliding Window
and size threshold to order objects into set classes. This method of choosing a frequency in conjunction with a size threshold value lead to some decent assumptions about various “classes” of objects based on their frequency and size class. As such, CSS and PSS perform quite well. We found that a maximum frequency of 128 was a good enough range [6]. Thus, in this research, we capped any frequency inputs to 128.

As stated before, CSS and PSS also incorporated object classes based on their size and their frequency as well. An object i belonged to a class j if and only if \( j = \log(S_i + 1) \). For NNPCR-2 we decided to divide objects into their size classes as well, as opposed to capping the threshold. This was done in hopes of pushing the neural network to generalize based on objects within their size classes. We then had to add a threshold to the size classes instead of the maximum object size. Since the size class is a logarithmic scale, the neural network has the ability to realize precise relationships with objects at lower sizes; between 2 to 1024 bytes there are 10 unique classes, while from 1024 to 2048, there are only 2 unique classes. This allows the neural networks to generalize about objects that are within class sizes, instead of specific sizes, which tends to be the case statistically as well. For this research, we chose a maximum size class of 22, which holds objects in the range \([2^{22}, 2^{23})\) bytes or up to 8 MB. Objects above 8 MB are essentially capped at that value and examined as if they were from the size 22 class.

### 3.1.2 Target Value and the Sliding Window

Now that we have set what the inputs will be for each pattern, we must now figure out what the neural network output will signify, while still keeping it as some classification problem dealing with the likelihood of a future request. However, in order for the network output to signify anything, we must know what the target output value will be during the training process.

Building on the aforementioned sliding window on a particular request, the main method is to use the sliding window length of information in the past to make judgments on what will occur in the sliding window length ahead of the request. Since the previous section dealt with the sliding window prior to the request, then clearly the target value must deal with information following the request.

Semantically then the output node of the neural network must then represent the likelihood that a request occurs within the sliding window length after a request or particular pattern. Thus, simply, the target value of the neural network is 1.0 when there is in fact a request to the same object within the SWL afterwards, and 0.0 otherwise. This is represented in Equation 6.

\[
\text{aget}_x = \begin{cases} 
1.0 & \text{if time to next request of the same object} \leq \text{SWL} \\
0.0 & \text{otherwise}
\end{cases}
\quad (6)
\]

Since the actual output of the neural network can also represent a probability, then anything that is above 0.5 can be classified as cacheable and anything below, as unlikely cacheable within the SWL. This is how we measure how the neural network correctly classifies a request. If we take the ratio of the number of patterns correctly classified as cacheable or unlikely cacheable to the total number of patterns, then we have the Correct Classification Ratio (CCR).

### 3.1.3 Frequency Values and the Sliding Window Length

When training the neural network, we estimated how the frequency values might be represented in a functional cache; we did not want to continually keep incrementing the frequency counter because of how the neural network was being trained. Assume the case where based on the SWL a target value of a particular request is set to 0. In an actual implementation of NNPCR-2, the corresponding object might be removed as a result, and thus, its frequency counter would be reset when it was removed from the cache. When it was seen again, its frequency counter would be reinitialized to 1.

Thus, we assume that if \( \Delta f \) is less than \( \text{SWL} \), then frequency of object \( x \) will be incremented by 1. Otherwise, the frequency count would be `decayed` by the number of sliding windows it is away from or set to 1, whichever is greater as shown in Equation 7. Note that if a previous request did not occur before, then frequency\((x)\) is set to 1.

\[
frequency(x) = \begin{cases} 
\max(frequency(x) + 1, \Delta f / \text{SWL}) & \text{if } \Delta f \leq \text{SWL,} \\
network \) otherwise
\end{cases}
\quad (7)
\]

The reason for the frequency decaying factor is that if the previous request was outside \( \text{SWL} \), then clearly the target value of that previous request would be 0.0, and thus, unlikely to remain in the cache (we assume that the expected value of the frequency count in this scenario is proportional to the number of sliding windows that separate the requests). Conversely, if the previous request was within \( \text{SWL} \), then chances are, it would remain in the cache and thus the frequency information would be retained fully.

### 4. Performance Metrics and Emulation Setup

Once our neural networks were trained and we verified our implementation in Squid, we ran several tests to measure the performance of how NNPCR-2 would fair against the other cache replacement strategies. We used a combination of a links file and our own edited version of `wget` [15], which did everything the same as `wget` except it did not write the data it received from the network to the file, to emulate the request streams recorded in the trace files. This was mainly done because we had very limited...
space restrictions and as well to decrease the training time by the amount of disk latency.

4.1 Performance Metrics

Performance metrics are designed to measure the three different goals a web cache should meet: decrease user perceived lag, decrease overall bandwidth, and decrease load on original servers. Hit-rate and byte hit-rate are by far the most common, and stable measurements of web cache performance as a whole.

- **Hit-rate.** This is simply the number of cache hits that occur to the total number of cacheable requests seen by the proxy. It’s important to realize that on average, 35-55% of the trace files we used were non-cacheable requests. Also important to note is that hit-rate is relatively a good measure of the user perceived lag.

- **Byte-hit Rate.** This metric is similar to hit-rate, except it emphasizes the total bytes saved by caching certain objects. As a result, byte hit rate is an indicator of the total bandwidth we are saving for the proxy cache and origin servers. Letting \( h_i \) be the total number of bytes saved from all cache hits that occur for an object \( i \) and \( r_i \) be the total number of bytes for all cacheable requests to object \( i \), and \( n \), the total number of unique objects seen in a request stream, then the byte-hit rate is:

\[
\frac{\sum_{i=1}^{n} h_i}{\sum_{i=1}^{n} r_i}
\]  

(8)

It is important to keep in mind that hit-rate and the byte-hit rate cannot be optimized for at the same time [1]. No cache replacement strategy can be deemed as the best because there is a tendency in request streams for smaller documents to be requested more often than larger ones due to the download time it takes to gather these objects. Strategies that optimize for hit-rate typically give preference to objects in a smaller size range, but in doing so tend to decrease byte-hit rate by giving less consideration to larger objects.

4.2 Trace Files

In our experiment, we used trace files, flat-files with recorded web requests, to test each replacement strategy. The trace files were provided by IRCache [14]. IRCache gathers their trace files and other data on web caching from several different proxy servers located around the United States. More than enough information is provided in these files to indicate which requests were cacheable.

Originally, the trace files were provided with data spanning only a day of recorded web requests. We strung seven consecutive trace files together to create a week long trace file from each proxy that the data came from. Once this was done, we then ‘cleaned’ the files to have only the cacheable requests. We also exchanged each unique URL that appeared in the file with a unique integer identifier so that test initiation time could be decreased as well.

Table 4 presents statistics about the three traces we used for this emulation. Non-cacheable requests were extracted from the files prior to our experiment as mentioned earlier. Each trace represented varying levels of temporal locality, spatial locality, total bandwidth and number of requests testing the various limits of the replacement strategies. For both UC-Test and NY-Test, we used trace files spanning from April 3rd to April 9th, 2008 and for the PA-Test trace file, only an older set from July 19th to July 25th, 2007 was available at the time of this research.

<table>
<thead>
<tr>
<th>Trace File Location</th>
<th>Urbana-Champaign, Illinois (UC-Test)</th>
<th>New York, New York (NY-Test)</th>
<th>Palo Alto, California (PA-Test)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Requests</td>
<td>2,223,555</td>
<td>1,767,052</td>
<td>519,232</td>
</tr>
<tr>
<td>Cacheable Requests</td>
<td>884,928 (39.79 %)</td>
<td>778,486 (44.06 %)</td>
<td>141,466 (27.24 %)</td>
</tr>
<tr>
<td>Total Bytes</td>
<td>181.96 GB</td>
<td>21.65 GB</td>
<td>8.28 GB</td>
</tr>
<tr>
<td>Cacheable Bytes</td>
<td>96.77 %</td>
<td>82.72 %</td>
<td>82.51 %</td>
</tr>
<tr>
<td>Unique Objects</td>
<td>1,050,160 (47.23 %)</td>
<td>853,433 (48.29 %)</td>
<td>269,476 (51.90 %)</td>
</tr>
<tr>
<td>Cacheable Objects</td>
<td>53.27 % of Above</td>
<td>54.02 % of Above</td>
<td>31.27 % of Above</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.3 Emulation Setup and Parameters

Once our trace files were parsed, we first created all of necessary web objects that we would need for testing on a separate server from the web proxy server. We used the size of the request and the URL to designate unique objects. Then for each object in each trace file, we created a unique file with the same number of bytes as recorded in the trace. This process was done by essentially just filling up the file with as many newlines (\"\n\") as there were bytes due to the fact that each character is a single byte. Once finished, we created a `wget` link file, which essentially is a flat file with a set of URLs that is requested in order one at a time.

There were two main issues with this approach, however. First, the files are requested one at a time, not asynchronously. Second, we could not control the timing of the requests so as to sync with the trace file durations, etc. This lead to our edited version of `wget` requesting the URLs as fast as it could on one thread, which leads to an entire week’s worth of web requests to being finished in about 11 hours of testing. Thus, certain characteristics of the request stream could not be replicated from the trace file into our emulation. Clearly, the recency values of requests/objects in our emulation did not sync up to what the recency values when the trace file was recorded. However, relative
characteristics such as frequency and size would of course prevail in either testing environment.

In an attempt to correct the problem of correlating recency values as they would have been seen in the trace file (since the trace file is clearly an accurate model of the practical environment), we multiplied all durations, $\Delta T_i$, by 25. This is equivalent to 1 week testing time / 25 = 6.72 emulated testing hours; 6.72 hours was close to the average total time that each link file completed within. In the implementation this means anytime that we needed to measure the amount of time that had passed since the last request occurred or the recency of an object, we would scale it by 25 times what was actually being reported.

Prior to running each test, we warmed up the cache with a smaller trace file from Boulder, Colorado. By using another trace file different from the others, we could guarantee that no files from that trace run would conflict with the other trace files. As a result, the cache would be filled by the time we started our simulation, putting our cache replacement strategies in effect immediately upon starting our tests. Therefore, all the results presented in Section 5 are the full results of the cache replacement strategy.

5. Emulation Results and Observations

We performed several different tests comparing different instances of NNPCR-2. We also compared NNPCR-2 with the other cache replacement strategies. Most results reported here have a standard deviation of error within 0.2 % for both metrics covered. Table 5 refers to the different parameters used for the various versions of NNPCR-2 tested. All NNPCR-2 tests used a max frequency of 128 and a size class of 22 (since we configured Squid to not cache objects larger than 8 MB). On the whole we found that the neural network trained with the 4 inputs and the UC testing file was the best instance of NNPCR-2 consistently on all trace files.

<table>
<thead>
<tr>
<th>Test Name</th>
<th>Structure</th>
<th>Training File</th>
<th>SWL</th>
</tr>
</thead>
<tbody>
<tr>
<td>NNPCR4-UC</td>
<td>4/10/10/1</td>
<td>UC</td>
<td>30 min</td>
</tr>
<tr>
<td>NNPCR4-NY</td>
<td>4/10/10/1</td>
<td>NY</td>
<td>30 min</td>
</tr>
<tr>
<td>NNPCR3-UC</td>
<td>3/10/10/1</td>
<td>UC</td>
<td>30 min</td>
</tr>
<tr>
<td>NNPCR3-NY</td>
<td>3/10/10/1</td>
<td>NY</td>
<td>30 min</td>
</tr>
<tr>
<td>NNPCR3-PA</td>
<td>3/3/3/1</td>
<td>PA</td>
<td>2 hours</td>
</tr>
</tbody>
</table>

Since we used the implementation of the heap class in Squid, M-Metric was implemented in a non-optimal way. The Squid implementation of the heap only updates the key upon a cache hit or when the object is added into the heap. It does not update again otherwise. As a result, many keys become deprecated rather quickly in M-Metric, and objects which had high values keep those same values until an object is either requested again or finally removed from the cache. In this implementation, M-Metric thus suffers greatly from cache pollution and its performance is severely hindered. However, M-Metric for this research really served as a lower bound to test NNPCR-2 against and as a way for us to validate our own implementation. Thus, we expected M-Metric to have the worst performance metrics in all testing simulations.

5.1 New York Test Results

Due to space limitations we will not report our tests for Palo Alto trace file. The New York test run produced steadier results than the Palo Alto test results. There were no large increases or decreases after the first 20,000 requests as seen in the Palo Alto test runs. Also notice, as shown in Table 4, that there were several times more requests in this test than in the PA-Test. The most interesting result here is how the hit rate seems to stabilize around 18% for the better strategies, but the byte-hit rate increases as the hit-rate decreases slightly over time. This demonstrates that the NY-Test request stream has large-sized requests coming through, and as Figures 3 and 4 show, both NNPCR4-UC and LFU-DA give precedence to the larger objects instead of the smaller objects.

![Figure 1: NY-Test Hit-Rate Results for different instances of NNPCR](image1)

![Figure 2: NY-Test Byte-hit Rate Results for different instances of NNPCR](image2)
NNPCR3-UC was the worst performance-wise for the different instances of NNPCR-2 as shown in Figures 1 and 2. The rest of the strategies performed so similarly that unless we zoomed in much further on the graph, it is almost impossible to tell the difference in performance in these instances. For consistency, we chose NNPCR4-UC to compare in Figures 3 and 4.

Again, we saw the same ranking in results as PA-Test. However, GDSF suffered a larger drop in byte-hit rate performance, barely doubling M-Metric’s own performance. Clearly, GDSF gave far more preference to smaller files due to its higher hit rate. However, in comparison to LFU-DA, LRU and NNPCR4-UC, they suffered a 3-4% hit rate drop in return for almost double the performance in byte-hit rate.

In almost all regards, this is a clear demonstration of intelligent decision making on the part of the replacement strategies — and most importantly, NNPCR-2 was able to pick up on this relationship and take advantage of it like LFU-DA and LRU.

5.2 Urbana-Champaign Test Results
The Urbana-Champaign test had the most stable results. It had a little over 100,000 more requests than NY-Test and almost nine times more than PA-Test. This test unlike the other two seemed to have coordinating byte-hit and hit rate performance changes. That is, there was a positive correlation between changes in hit rate to changes in byte-hit rate.
Figures 5 and 6 compare the different instances of NNPCR-2. To almost no surprise again, however, NNPCRC3-UC did the worst, with the other NNPCR instances not having much of a deviation from each other. This demonstrates the ability of NNPCR-2 to be able to generalize quite well if tuned correctly during training with the proper variables. This may also be a hint as to how request streams are statistically.

Figures 7 and 8 show that NNPCRC4-UC did better than LFU-DA some of the time. From the Figures, we noticed, unlike NNPCR-2, the severe penalty in GDSF’s ability to make a balanced decision between hit rate and byte-hit rate.

6. Conclusions

We have suggested many improvements to NNPCR and dubbed it NNPCR-2. Mainly we added an aging factor to deprecate objects from the cache in order to prevent cache pollution; this greatly improved its performance and allowed it to run in an actual implementation for longer periods of time. We implemented and tested NNPCR-2 in a real, practical web proxy cache framework. Clearly NNPCR-2 was able to make better, balanced decisions than LRU and GDSF and had similar performance with LFU-DA. From our results, we see that NNPCR-2 is absolutely a plausible proxy cache replacement strategy.

Although we trained on a day’s worth of patterns and emulated a week’s worth of requests NNPCR-2 can handle even larger training and testing sets. Once trained, NNPCR-2 requires no additional parameter tuning and handles changes in the request stream characteristics quite well. It is absolutely possible to provide default neural network structures and already trained configurations of NNPCR-2 to end-users and system administrators in ready-to-use configurations.

10. References