Disk Cache Replacement Policies for Network Fileservers

Darryl L. Willick, Derek L. Eager, and Richard B. Bunt

Department of Computational Science
University of Saskatchewan
Saskatoon, Canada S7N OWO

October 13, 1992

Abstract

Trace driven simulations are used to study the performance of several disk cache replacement policies for network fileservers. It is shown that locality based approaches, such as the common Least Recently Used (LRU) policy, which are known to work well on standalone disked workstations and at client workstations in distributed systems, are inappropriate at a fileserver. Quite simple frequency based approaches do better. More sophisticated frequency based policies (e.g., that take into account the file type) may offer additional improvements.

1 Introduction

Distributed systems consisting of workstations and shared fileservers typically have main memory disk block caches at both the workstations and the fileservers. These caches form a two-level hierarchy, in which file I/O requests (making up a "disk reference stream") generated by applications running on the clients may be satisfied at the local cache, or if not, then possibly at the fileserver cache. These caches improve system performance by reducing the frequency with which costly disk operations are performed, and, in the case of the caches at the workstations, the frequency of requests to the fileservers.

In this paper, we study the suitability of various *replacement policies* for fileserver caches. The replacement policy specifies which disk block should be removed when a new block must be entered into an already full cache, and should be chosen so as to ensure that blocks likely to be referenced in the near future are retained in the cache. Disk block caches have almost universally employed the "Least Recently Used" (LRU) policy, in which the cache block that is least recently used is replaced. This policy is justified by the temporal locality that commonly exists in file I/O request streams, and studies of client caches have shown it to work well [2]. Nonetheless, in spite of the universal adoption of LRU, we feel that there are two compelling motivations for our work.

First, fileserver caches differ, both in comparison to the workstation caches, and in comparison to the well-studied caches in centralized environments, with respect to the characteristics of the disk reference streams that they handle. These reference streams differ greatly from those generated by the applications themselves, as all references satisfied by client caches have been removed. This "filtering" of the reference stream destroys much of its temporal locality, as was suggested by previous work [5], and as we show in this paper.

Second, the choice of replacement policy is one of the most critical cache design issues. Some have argued that workstation caches are now becoming so large that they can hold the complete disk block "working sets" of their users, and thus that the ability of a fileserver cache to hold portions of these working sets is irrelevant [9]. While this may be the case now (at least in many environments), we claim that it is unlikely to be the case in the future. In particular, the vastly increased file sizes intrinsic to the rapidly emerging multimedia applications [6, 10] will almost certainly overwhelm workstation caches. Thus, the choice of replacement policy at fileserver (as well as at workstation) caches will continue to be an important issue.

This paper uses trace driven simulations to evaluate several possible server cache replacement policies. Through performance comparisons it is shown that locality based algorithms such as LRU are inappropriate at a fileserver. It is also shown that quite simple (and "untuned") frequency based algorithms do better. More sophisticated frequency based policies (e.g., that take into account the file type) may offer additional improvements.

The remainder of this paper is organized as follows. Section 2 describes several disk cache replacement policies, and in particular, the policies evaluated in this study. Section 3 describes the experimental methodology. Section 4 presents and discusses the simulation results. Finally, Section 5 concludes the paper.

2 Disk Cache Replacement Policies

There are a number of disk cache replacement policies which could be used at a fileserver cache. This study investigates the performance of five of these: least recently used (LRU), least frequency used (LFU), frequency based replacement (FBR), MIN (the optimal strategy), and RAND (random replacement).

LRU replaces the block in the cache which has not been used for the longest period of time. The basic premise behind the LRU algorithm, and the reason for its success, is that blocks which have been referenced in the recent past will likely be referenced again in the near future. This property, known as *temporal locality*, is known to be a characteristic of memory reference patterns of processors, page reference patterns in virtual memory systems, and to a lessor extent, disk block access patterns of applications running on workstations [5, 8]. The basic question that this study attempts to address is: "Does temporal locality still exist in the disk block reference stream seen by a *fileserver* to the extent that LRU can successfully take advantage of it, or is it disrupted sufficiently by the presence of client caches so that LRU is no longer a good predictor of which blocks will (and will not) be referenced again?"

LFU requires that a reference count be maintained for each block in the cache. When a replacement is necessary, LFU chooses the block which has the lowest reference count. (In our implementation, a tie is resolved by replacing the least recently used of those blocks). The motivation for LFU and other frequency based algorithms is that the reference count can be used as an estimate of the probability of a block being referenced. Blocks with high counts are more likely to be referenced than blocks with low counts. In the normal single level cache (such as a single multitasking workstation with local disks), LFU is known to work poorly (and results to be described later confirm this). One problem with the use of frequency counts is that certain blocks may build up high reference counts and never be replaced. Although this may be appropriate in some instances, it is more likely that the high probability of referencing such a block does not hold over a long period of time but rather only over a relatively short period. A solution to this issue is to introduce some form of reference count "aging" (or decay). The approach used here is identical to that described in [8]. The average reference count (over all blocks currently in the cache) is maintained dynamically. Whenever this average count exceeds some predetermined maximum value, A_{max} (a parameter to the algorithm), every reference count, C, is reduced to $\lceil C/2 \rceil$.

Another problem with LFU is that it suffers from the effects of temporal locality. Although a block may be relatively infrequently referenced overall, it may experience short bursts of re-references which build up its reference count to a misleadingly high value. This problem (which is different than the previous) has been the major failing of the LFU algorithm for cache replacement policies in traditional environments. In the distributed environment, however, this problem may be eliminated automatically. It is probable that the temporal locality is "filtered" out by the caches at the client workstations. In fact, results show that this does seem to be the case.

Preliminary experiments showed that LFU, as described, also suffers from the fact that many write requests issued by applications only write data to part of a full block (a partial block write). When such a write request is issued, the following must occur. First, the client cache is searched for the requested block. If it is found (a "client hit"), then the contents are modified by simply merging the new partial write with the old contents. This new block is then written to the server (write-through is assumed). If the requested block is not found at the client cache (a "client miss"), however, it must first be *"read"* from the server so that the partial write may be merged with the old contents. Once this is done, the block must be written back to the server. Therefore, a partial write request which "misses" at the client actually generates a read request immediately followed by a write request at the fileserver cache. This, in turn, results in the block's reference count

- For a write miss:
  - insert the block into the cache
  - set its reference count to one
  - set a FLAG for this block to TRUE
- For a write hit:
  - don't modify the reference count or the FLAG for this block
- For a read miss:
  - insert the block into the cache
  - set the block's reference count to one
  - set a FLAG for this block to FALSE
- For a read hit:
  - IF the FLAG for this block is FALSE,
    THEN increment this block's reference count,
    ELSE set its reference count to one and its FLAG to FALSE

Figure 1: Modified Reference Counting

at the server being incremented *twice* for a single write operation. This double counting causes a misleadingly high reference count for some blocks in the server cache and affects the performance of the LFU policy. There is a simple solution to this problem: simply do not increment reference counts on writes. Simulations showed that this modification greatly improves the performance of the LFU algorithm at the fileserver. Therefore, this modification is assumed in the LFU algorithm studied in the remainder of this paper. (The algorithm for incrementing reference counts is given more precisely in Figure 1.)

Robinson, et. al [8] propose a new policy which they call Frequency Based Replacement (FBR). Although they study it within the context of a standalone system, it could also be applied to a fileserver. FBR is a hybrid replacement policy, attempting to capture the benefits of both LRU and LFU without the associated drawbacks. FBR maintains the LRU ordering of all blocks in the cache, but the replacement decision is primarily based upon the frequency count. To accomplish this, FBR divides the cache into three partitions (see Figure 2): a new partition, a middle partition, and an old partition. The sizes of these partitions are specified by two parameters to the model. F_{new} indicates the percentage of the total number of cache blocks which are contained in the new section (the most recently used (MRU) end), while F_{old} indicates the percentage of blocks contained in the old section (the least recently used (LRU) end). The middle section consists of those blocks not in either the new or the old section. When a reference occurs to a block in the new section, its reference count is NOT incremented. This is done to explicitly "factor out" temporal locality which is the primary reason for the past failure of frequency based algorithms. References to the middle and old sections do cause the reference counts to be incremented. When a block must be chosen for replacement, FBR chooses the block with the lowest reference count,

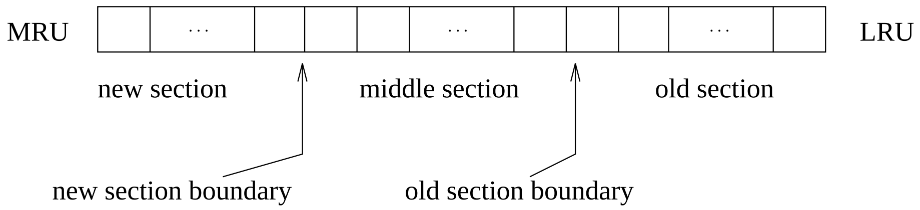

Figure 2: Three Sections of The FBR Cache

but only among those blocks that are in the *old* section of the LRU ordered stack (ties are resolved by choosing the least recently used of those blocks). This is done because the blocks in the new and middle sections have not had enough time to build up their reference counts. The FBR algorithm uses the reference count aging scheme described for the LFU algorithm. Preliminary experiments showed that when FBR is used at the fileserver cache it benefits from the same modification to the reference counting scheme as previously described for the LFU algorithm. Therefore, this modification is assumed in the remainder of this paper (see Figure 1). It should be noted that by choosing the parameters appropriately ($F_{new} = 0$ and $F_{old} = 100$) the FBR algorithm becomes the LFU policy.

MIN is the theoretically optimal replacement policy. That is, it provides the lowest miss ratio of all possible algorithms. The algorithm is simply, "Replace the block which will not be used for the longest period of time." Unfortunately, the optimal replacement policy is impossible to implement since it requires future knowledge of the disk reference string. The performance of this policy is useful for our study, however, since it provides an upper bound on the performance that is attainable, and thus provides some feel for the "goodness" of the other policies.

RAND chooses among all blocks in the cache with equal probability. Intuitively, RAND seems appealing in this context if the client caches are filtering out all of the locality characteristics from their reference streams. For RAND to work well, however, it would also be required that all disk blocks are accessed with equal probability. This is not likely to be the case. Even so, RAND is a useful policy to study since it provides yet another basis against which to evaluate policies which attempt to make "intelligent, informed" replacement decisions. If they cannot perform better than RAND, then there is almost certainly some underlying flaw in the decision strategy. In effect, RAND, provides a kind of lower bound on performance. That is, there is no reason to use any policy which performs worse than RAND.

Korner [3] investigates an alternative cache management policy for distributed fileservers. His algorithm chooses among different replacement strategies (LRU, MRU, or no caching) for different classes of blocks depending upon the anticipated reference patterns. Performance improvements of up to 340% were observed over that of LRU alone. This algorithm is not explicitly evaluated in this study. However, by evaluating the LRU algorithm, this paper demonstrates that, possibly, LRU should not be one of the three strategies that Korner's algorithm chooses from.

3 Experimental Methodology

The approach taken in this study was to gather real disk block reference streams generated under realistic workloads and then use these traces to drive simulation models implementing the replacement policies to be compared. This section begins with details of the trace collection process. The simulation model is described next, followed by some comments on the simulation methodology.

3.1 Trace Collection

In order to obtain the reference traces necessary to drive a simulator modifications were made to the HP-UX 8.0 operating system kernel to record an event for each cache block lookup operation (a call to the internal kernel routine getblk() [1, 4]). Among other things, each getblk() event in a trace records the device id and inode number of the file referenced, as well as the logical block number. These (device id, inode #, logical block #) triples can be used to represent a unique disk block within the cache simulator with one exception. Since files may be removed and created during the span of a tracing period, some inode numbers may be reassigned to new files. If this situation were not properly accounted for, the simulator would predict a higher hit ratio than appropriate. To properly account for this effect, the gathered traces also contain "remove" events produced by tracing the "unlink" system call. These events record the device and inode number of the file being removed, and therefore provide the simulator with enough information to invalidate the appropriate blocks upon the removal of a file.

In order to simulate both the client and server caches as they work together, traces were gathered on the client workstations rather than on the server. The reference stream seen by the server cache is thus derived during the simulation, and is basically the set of "misses" which occur at the client cache. (There are other events as well, such as the "remove" events and write events which must propagate to the server even if they "hit" at the client.)

Eight traces were gathered for the purposes of this study. The type of activity traced ranged from large compilations (over 450 source files comprising more than 300,000 lines of C source code), text editing/processing, to the execution of large file intensive graphics/image processing applications. It was felt that this set of traces represents the range of file access activity typical within research environments. Table 1 summarizes the eight traces.

3.2 Simulation Model

A trace-driven simulation was written to simulate the operation of both the client cache and the server cache. Since we are interested only in studying alternative replacement policies at the server cache, the client cache always uses LRU replacement. The server replacement policy can be set to any of the five policies described in Section 2.

Trace	Duration (hours)	Number of reads	Number of writes	Dominant Activity
1	3.80	111947	31661	Kernel compiling from source code
2	1.57	5871	1710	Program development (edit, compile, debug)
3	0.78	3570	743	Program development (edit, compile, debug)
4	1.20	3734	447	Document editing and processing
5	1.11	2595	431	Document editing and processing
6	1.69	11304	5498	Program development (edit, compile, debug)
7	0.52	11301	4372	Graphics/Image processing
8	2.03	40805	15654	Graphics/Image processing

Table 1: Trace Characteristics

The simulator is given the reference stream (trace) as it would be presented to the client's cache (see Section 3.1). Reading each event in turn, the simulator first simulates the actions that it would cause at the client cache, followed by those that would be caused, if any, at the server cache. (It is assumed that there is no local activity at the fileserver.) The actions caused by each type of event are now described in further detail.

For a read operation, the client cache is searched first. If the requested block is found (a client cache read hit) then no action propagates to the server, and the next event is read from the trace. Otherwise, a read miss is recorded at the client cache and a read operation will be propagated to the server. The server cache will be searched for the block, and if it is found, there is a server cache read hit. If the block is not found, there is a server cache read miss and the block must be read from the server's disk and placed into the server cache. If there is no empty cache block to use, then an existing block is first removed according to the server cache replacement policy. Whether or not the block was found in the server cache, it is then returned to the client where it is placed into the client cache (possibly causing a replacement there according to the LRU policy).

For write operations, a write-through write policy is assumed. Although policies such as delayed-write or copy-back might be more common in practise, previous studies have found that the choice of write policy does not significantly alter qualitative results [8] (preliminary experiments also confirmed this). In our simulation model, a write operation causes the following actions. The client cache is searched for the requested block. If it is found (a client cache write hit), it is moved to the top of the LRU stack (as with reads). If it is not found (a client cache write miss), it is placed into the client cache, causing a replacement if necessary. Whether or not the written block was found at the client cache, the write will propagate to the server (because of the write-through assumption). The server cache is searched for the block, and if it is found (a server write hit), the cache will be updated depending on the replacement policy's needs (e.g., moved to the top of the stack if LRU or FBR). If the block is not found (a server write miss), then it is placed into the server cache, causing an existing block to be removed (if there are no empty blocks) according to the server cache replacement policy. Because of the write-through assumption, the write will also propagate to the physical disk whether or not it was a hit or a miss at the server cache.

The above procedure ensures that after a write operation, the client cache, server cache, and server disk all have up-to-date copies of the modified block. It should also be noted that with write-through there is no possibility to save disk operations on writes (they always cause a physical disk write). Therefore, the only metric of interest to this study is the READ hit and miss ratios. These are the only metrics reported in the remainder of this paper.

As described in Section 3.1, the disk block reference traces contain "remove" events as well as normal read and write operations. This is to account for the fact that the inode numbers in the (device id, inode #, logical block #) triples used by the simulator to identify blocks may actually be reassigned if a file is removed and a new file is created during the same tracing period. The "remove" events in the trace simply indicate which (device, inode #) pair corresponds to a removed file. Upon reading such an event, the simulator simply marks as "invalid" any blocks in both the client and the server caches which match this (device id, inode #) pair. Any ensuing read or write operations involving this pair will thus not match any blocks which have been invalidated. This is appropriate since these invalidated blocks actually belonged to the removed file and thus probably do not correspond to the same physical disk block. This technique should provide a level of accuracy sufficient for the purposes of this study.

3.3 Simulation Methodology

After some initial experimentation it was concluded that there was little difference in the qualitative nature of the results as the number of clients was increased from one. Therefore, this paper restricts its analysis to single client, single server simulations.

For each trace collected, simulations were run with each of the five replacement policies, with the client caches ranging from 0 to 200 blocks, and the server cache size ranging from 50 to 800 blocks. In the HP-UX operating system a block is 8K in size, and therefore these choices of cache sizes represent client caches up to 1.6 MB and server caches up to 6.4 MB. Note that with a client cache size of zero, the results correspond to what would be obtained were the workload running directly on a disked workstation using the particular replacement policy (i.e., a traditional standalone machine). These results (see Figure 3) are included to illustrate how much effect a first level client cache has on the reference stream seen by a fileserver and to show that LFU is completely inappropriate in a standalone environment but that it performs considerably better when applied to the (second level) fileserver cache.

Two choices of parameters were simulated for the FBR policy. FBR-1 sets $F_{new} = 0.25$ meaning that the new section of the cache contains the top 25 percent of the LRU stack, while $F_{old} = 60$ meaning that the old section contains the bottom 60 percent of the LRU stack. These parameter choices are consistent with [8] where they were obtained through experimentation. The second FBR policy simulated, FBR-2, sets $F_{new} = 25$ and $F_{old} = 40$. For both of the FBR policies and the LFU policy also, A_{max} was set to 100 (again from [8]).

4 Simulation Results

This section presents and discusses the simulation results obtained.

Figure 3(a) and Figure 4 show the average server miss ratio for the five policies simulated as a function of the server cache size. Graphs are presented for each of the five client cache sizes. Each point is obtained by taking the average over all traces (weighted by the number of read operations in the trace).

By examining these graphs, several observations can be made:

- Figure 3(a) examines performance for a single level cache (i.e., at the client, or at a standalone workstation). Clearly the LFU policy is inappropriate in such an environment, as expected. The FBR algorithm does correct the problems caused by temporal locality, making it comparable to LRU. It should also be noted from this graph how close the LRU, FBR, and even the RAND policies are to the optimal miss ratio (MIN). There actually is little room for improvement in this configuration.

- Figure 4(a) shows that with even a small cache at the client, there is a dramatic performance degradation for all of the algorithms *except* for LFU. LFU actually improves in most cases, making it a much more competitive policy in this configuration.

- As the client cache size increases further (Figure 4(b)-(d)), the performance of all of the algorithms degrades, including MIN. This indicates that the potential usefulness of the server cache decreases considerably as the client cache size increases.

- LRU suffers the most as the client cache size is increased, making it the worst policy in many cases (even worse than RAND in several cases). As suspected, the client cache is apparently removing from the reference stream the feature that makes LRU work well – the temporal locality.

- LFU outperforms LRU in all cases where there is a client cache except where the client cache is very small and the server cache is larger (Figure 4(a)). It seems that in this case the client cache is not large enough to remove most of the temporal locality.

- FBR's initial performance advantage over LFU for small client caches is probably because of its *new section* which is able to filter out the excess temporal locality aspects from the reference stream which the small client caches do not already remove.

- As the client cache size increases, the performance of LFU degrades at a slower rate than that of LRU and FBR, so that by figure 4(c), LFU's performance is virtually identical to FBR's and considerably better than LRU's.

In order to gain a better understanding of the poor LRU performance and relatively better performance of the frequency based policies at the fileserver, one of the traces (trace 1) is considered in more detail. Figure 3(b) and Figure 5 show the server miss ratio as a function of server cache size for trace 1 only. Note how the observations listed above hold for this figure as well.

To investigate how the characteristics of the reference stream seen at the fileserver change, and to understand why the LRU policy suffers more than the other policies as the client cache size increases, LRU stack depth histograms for trace 1 are shown in Figure 3(c) and Figure 6. These histograms are generated by running the simulation with the specified client cache size and an infinite server cache. Each bar of the histogram indicates how many "hits" occur at each LRU stack position. (To be consistent with previous results, only *read* hits are recorded.) Figure 3(c) shows the LRU stack depth histogram with no client cache. This is equivalent to what the stack histogram would look like on a standalone, disked workstation. Obviously, there is much temporal locality in the original trace. This is the basic assumption of LRU – that blocks referenced in the recent past are likely to be referenced again in the near future. This is evident by the large number of hits at the most recently used end of the stack. (In fact, the histogram had to be cut off for presentation purposes. The actual number of hits to the most recently used location (stack depth of one) is more than 30,000). This graph clearly illustrates why LRU works so well at the client caches or on standalone disked workstations. From Figures 6 we can see that the locality is, as expected, filtered out by client caches. This is evident by observing that the location of the hits at the server becomes more and more random as the client cache is increased in size. Note that the total number of hits decreases considerably as the client cache size is increased. This is because with larger client caches there are simply fewer references which are not satisfied at the client and that must propagate to the server. The gap with virtually no hits which occurs at the most recently used end of these figures deserves further attention. This gap appears because a block is moved into the most recently used position of the server cache only as a result of a miss at the client. This means that it is also moved to the most recently used end of the client cache. The server will not see any references to this block again until it has "fallen" out of the client cache. The chance that the block will still be near the most recently used end of the server cache is very small.

A similar technique was used to gain further insight into the LFU algorithm. Figure 3(d) and Figure 7 show LFU stack depth histograms for trace 1 at the server for various client cache sizes. The bar at stack depth equal to one indicates how many hits involved the most frequently used block currently in the cache; the bar at stack depth equal to two indicates the number of hits to the second from the most frequently used block (i.e., there was one block with a higher reference count in the cache); and so on for larger stack depths. From Figure 3(d) it can be seen that the reference count is not a very good predictor of which blocks will be used in the future for a standalone workstation or at the client cache. Although there is a spike at the most frequently used end of the histogram, many of the hits are spread out over a very large range (note that the x-axis in Figure 3(d) goes up to 3000). Also, there are actually only 4100 hits at this spike (the y-axis is cut off for clarity). This is considerably smaller than the 30000 hits at the most recently used position in Figure 3(c). These observations are consistent with the poor performance of the LFU algorithm on a standalone system as illustrated in Figure 3(a). Figure 7 shows that as a client cache is introduced before the server there is a dramatic change. In these graphs most of the hits are concentrated very close to the most frequently used end of the histogram (note the x-axis only goes to 1000), indicating that at the fileserver, the reference count becomes a reasonably good predictor of which blocks will be used in the future. Comparing these graphs with the corresponding LRU stack depth histograms (Figure 6), shows that, in fact, the concentration towards the most frequently used end is greater than the concentration of hits to the most recently used end of the LRU stack depth histograms. This is especially true for larger client cache sizes (Figure 8 shows these regions in greater detail). Note also that the "gap" at the low stack depth end in the LRU stack histograms is not present in the LFU histograms.

In an attempt to determine how *good* the replacement decisions of LRU and LFU are, their replacement decisions at the server cache were compared to those made by the optimal (MIN) policy.

First, for each replacement made by MIN, the position that the replaced block would be in if the cache were maintained in LRU order was used to build a histogram. The resulting histogram indicates how often MIN replaced a block from each LRU stack position (i.e., how many times it replaced the most recently used element, the second most recently used element, etc., up to the number of times it replaced the least recently used element). The shape of this histogram can be used to make a statement about the appropriateness of LRU under the simulated reference stream at the server. Since the LRU policy always replaces the least recently used element, we would expect that when MIN is making most of its replacements from the least recently used end of the stack, then the LRU policy would perform well. On the other hand, when MIN makes most of its replacements from the most recently used end of the cache or they are scattered randomly over all positions, the LRU policy would be inappropriate. To be fair to LRU, the MIN algorithm was tuned slightly so that if it finds that two or more blocks may be replaced with the same end result, then the least recently used of these blocks is chosen (this occurs only if there are blocks in the cache which never appear in the future reference stream again).

A similar technique was used to evaluate the appropriateness of the LFU policy. Here, the position of the replaced block when the cache is sorted by frequency count is used to build the histogram (with ties resolved by the LRU order, and reference count aging implemented identical to the LFU and FBR policies as described in Section 2). If the LFU policy is appropriate, one would expect MIN to be replacing the least frequently used (or close to least frequently used) elements most often. If LFU is not appropriate, then MIN would be replacing more elements from the most frequently used end of the sorted list or replacements would be scattered randomly. Similar to the LRU case, MIN was tuned slightly while generating these histograms so that if two or more blocks could be replaced with the same end result, then MIN chooses the least frequently used of these.

Selected histograms resulting from the above procedure (for specific client cache and server cache sizes) and using trace 1 are presented in Figures 9 and 10. Figure 9 shows the MIN replacement histograms for the case where the client cache is 150 blocks and the server cache is 400 blocks. (This corresponds to the server cache = 400 blocks points of Figure 4(c).) Examination of Figure 9(a) shows that frequency count seems to be a reasonably good predictor of which blocks should be replaced. This is evident by noting how the majority of replacement decisions made by MIN involve blocks with relatively low reference counts. On the other hand, Figure 9(b) shows that there is much less correspondence between the optimal block to replace and its LRU stack position (evident by the more uniform appearance of the histogram). This suggests that LRU is not a very good predictor of which blocks should be replaced from the server cache. Figure 10 shows similar trends for the case where the client cache is 200 blocks and the server cache is 400 blocks.

Examination of such histograms for the other points of Figure 4 shows that these results are fairly consistent (due to space limitations, the entire set of 60 graphs cannot be shown). In conclusion, it is apparent that LRU stack position is not a very good predictor of which blocks should be replaced from the server cache. The frequency count seems to be a better predictor.

5 Conclusions

The goal of this research was to investigate the suitability of various cache management strategies in the context of a network fileserver. Five policies were considered in total: LRU, LFU, FBR, MIN, and RAND.

Custom instrumentation of the HP-UX operating system kernel made it possible to gather disk block reference traces of real workload. These were used to drive a cache simulator which incorporated both client and server disk caches.

Our simulation results suggest that LRU is not a very good replacement policy for a fileserver cache. With the traces gathered for this study it was found that the average performance of the FBR algorithm is usually better than (or at least equal to) that of LRU as long as there is a some disk caching at the client. Even more surprising is that the simple LFU algorithm outperforms LRU in most cases, except where the client cache is very small and the server cache is larger. Studies of the stack depth histograms show that the temporal locality, upon which the success of LRU depends, is removed as the client caches increase in size. In addition, comparisons of the replacement decisions made by the optimal MIN policy at the fileserver with those that LRU would make show that there is not a strong correspondence. On the other hand, the correspondence between MIN's replacement decisions and frequency based approaches such as LFU are much stronger.

Although the FBR algorithm outperforms LFU for small client cache sizes, as the client cache size increases LFU performance becomes virtually identical to FBR.

There are several extensions to this work which are immediately obvious. First, this study only considered simulations with a single client and the fileserver. Although some preliminary experiments with multiple client simulations seem to suggest that the results reported here still hold, further in-depth studies should be performed. In particular, the effect of file sharing among clients on the performance of the server cache replacement policy should be investigated; it is likely to make the frequency based policies even more attractive. Second, although the traces gathered for this study are representative of the typical workload in our installation they are fairly short. Future work would benefit from larger traces (measured in days rather than hours) and from traces gathered in industrial environments. Third, more sophisticated fileserver cache management strategies such as Korner's [3], but with an underlying frequency based approach rather than an LRU approach would be worthwhile investigating.

6 Acknowledgements

The authors would like to thank Michael Molloy, from Hewlett-Packard, for providing technical assistance and several tools used to instrument the HP-UX 8.0 operating system. Kevin Froese did a great job of gathering the traces. Funding for this research was provided, in part, by the Natural Sciences and Engineering Research Council of Canada (NSERC), through Research Grants OGP0000264 and OGP0003707 and Strategic Grant STR0045250.

References

[1] Bach, M. J., *The Design of the UNIX Operating System,* Prentice-Hall, 1986.

[2] Baker, M. G., Hartman, J. H., Kupfer, M. D., Shirriff, K. W., and Ousterhout, J. K., "Measurements of a Distributed File System," *Proceedings of the 13th ACM Symposium on Operating System Principles,* October 1991, pp. 198-212.

[3] Korner, K., "Intelligent Caching for Remote File Service," *ICDCS,* May 1990, pp. 220-226.

[4] Leffler, S. J., McKusick, M. K., Karels, M. J., and Quarterman, J. S., *The Design and Implementation of the 4.3 BSD UNIX Operating System,* Addison Wesley, 1989.

[5] Makaroff, D. J., and Eager, D. L., "Disk Cache Performance for Distributed Systems," *ICDCS,* May 1990, pp. 212-219.

[6] Muller, K. and Pasquale, J., "A High Performance Multi-Structured File System Design," *Proceedings of the 13th ACM Symposium on Operating System Principles,* October 1991, pp. 56-67.

[7] Ousterhout, J. K., Da Costa, H., Harrison, D., Kunze, J. A., Kupfer, M. and Thompson, J. G., "A Trace-Driven Analysis of the UNIX 4.2 BSD File System," *Proceedings of the 10th ACM Symposium on Operating System Principles,* December 1985, pp. 15-24.

[8] Robinson, J. T., and Devarakonda, M. V., "Data Cache Management Using Frequency-Based Replacement," *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* May 1990, pp. 134-142.

[9] Rosenblum, M., and Ousterhout, J., K., "The Design and Implementation of a Log-Structured File System," *Proceedings of the 13th ACM Symposium on Operating System Principles,* October 1991, pp. 1-15.

[10] Venkat Rangan, P., and Harrick, M. Vin, "Designing File Systems for Digital Video and Audio," *Proceedings of the 13th ACM Symposium on Operating System Principles,* October 1991, pp. 81-94.

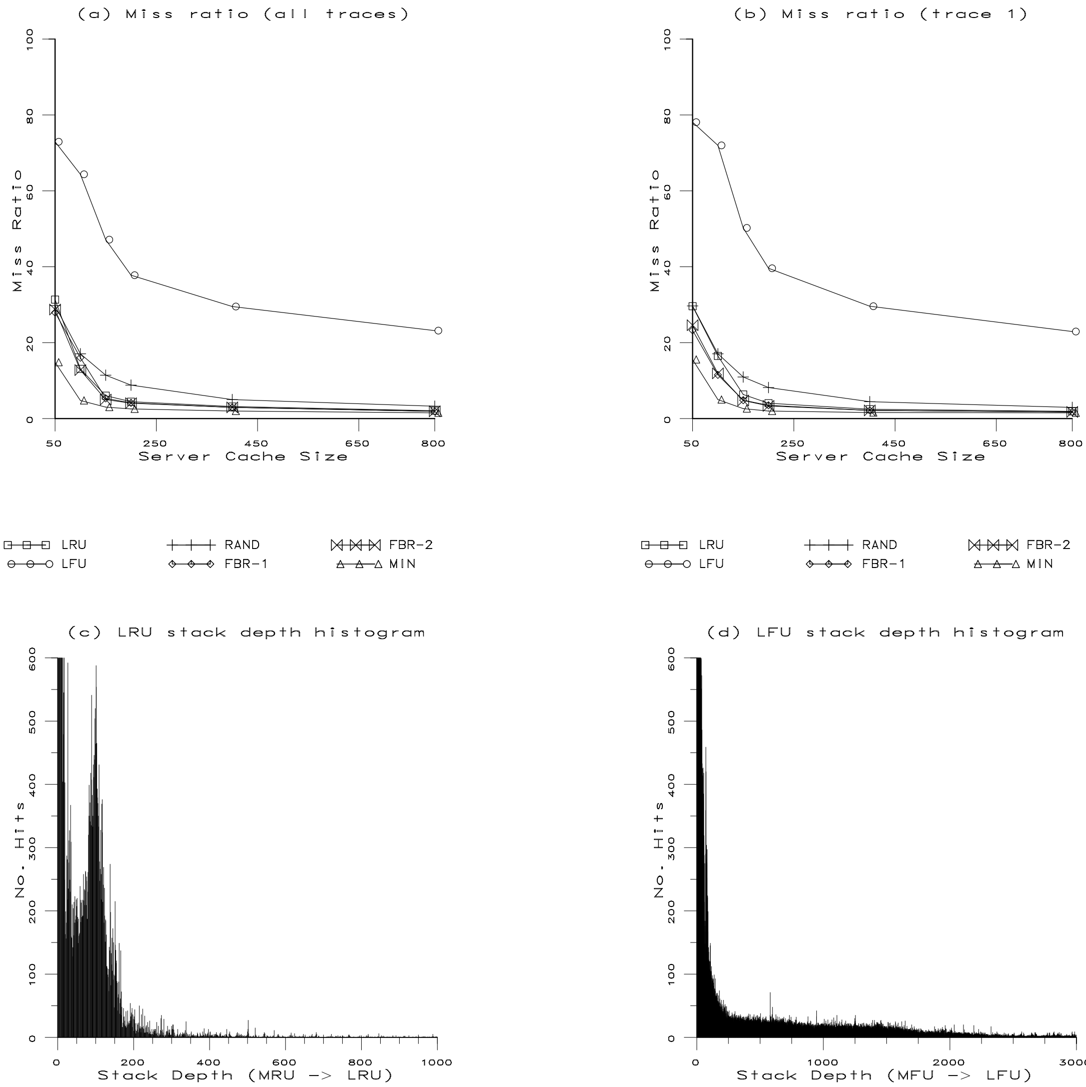

Figure 3: Conventional cache behaviour (i.e., on standalone system, or at a client cache). (Generated with the same simulator by setting the client cache size equal to zero and measuring the resulting server cache performance.)

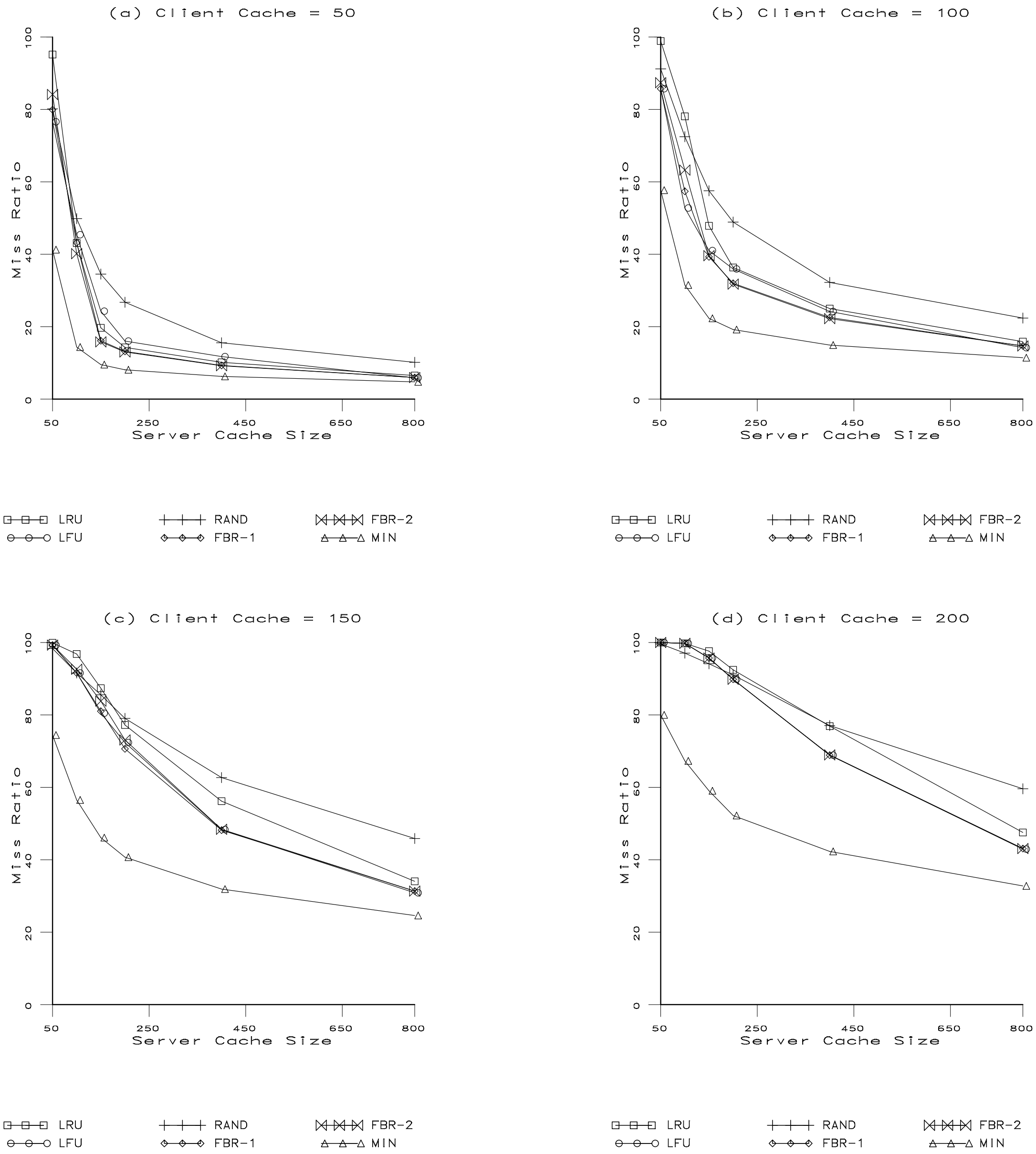

Figure 4: Server cache miss ratio as a function of the server cache size, averaged over all traces. Each graph is the result of a different client cache size.

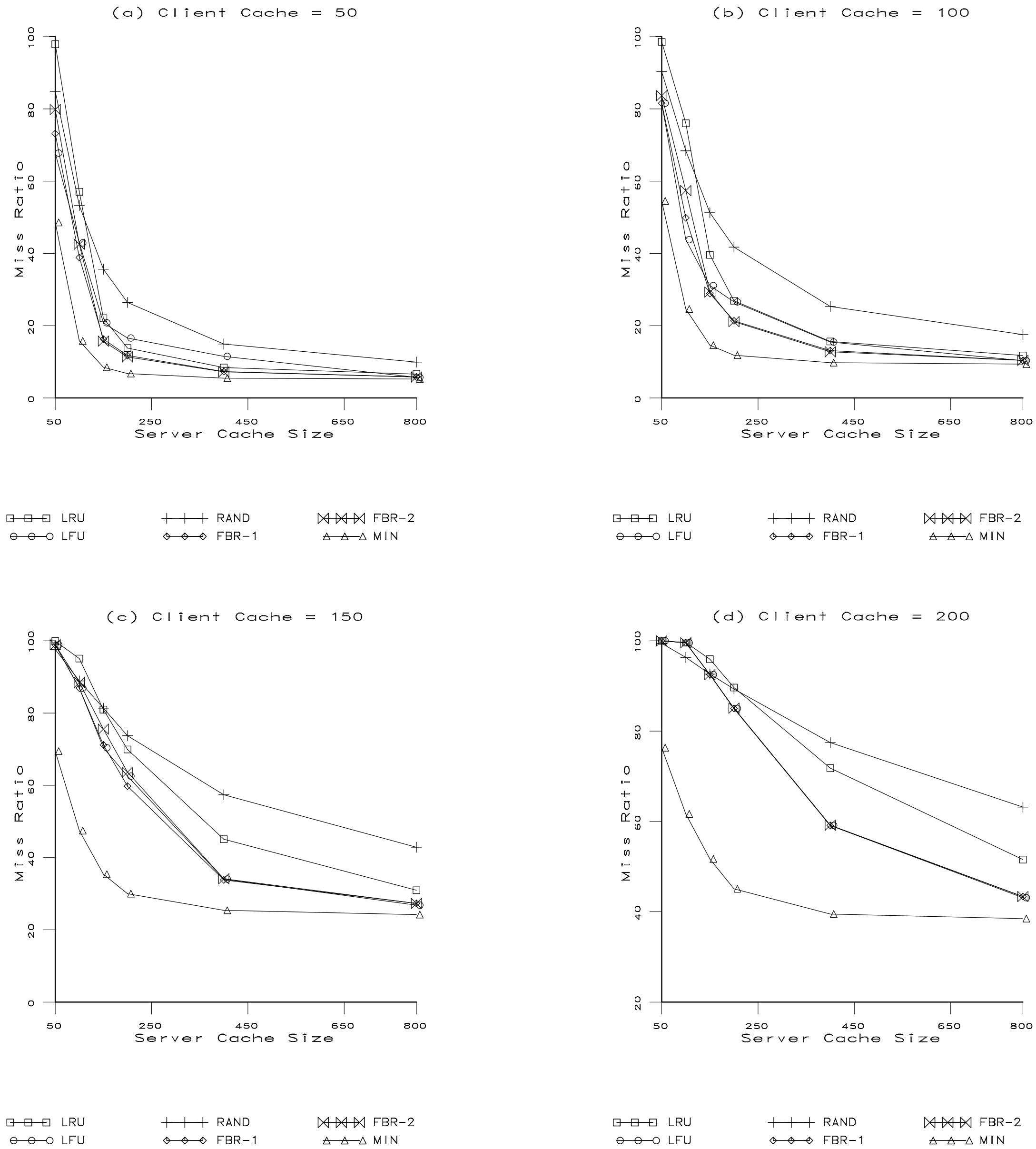

Figure 5: Server cache miss ratio as a function of the server cache size, for trace one only.

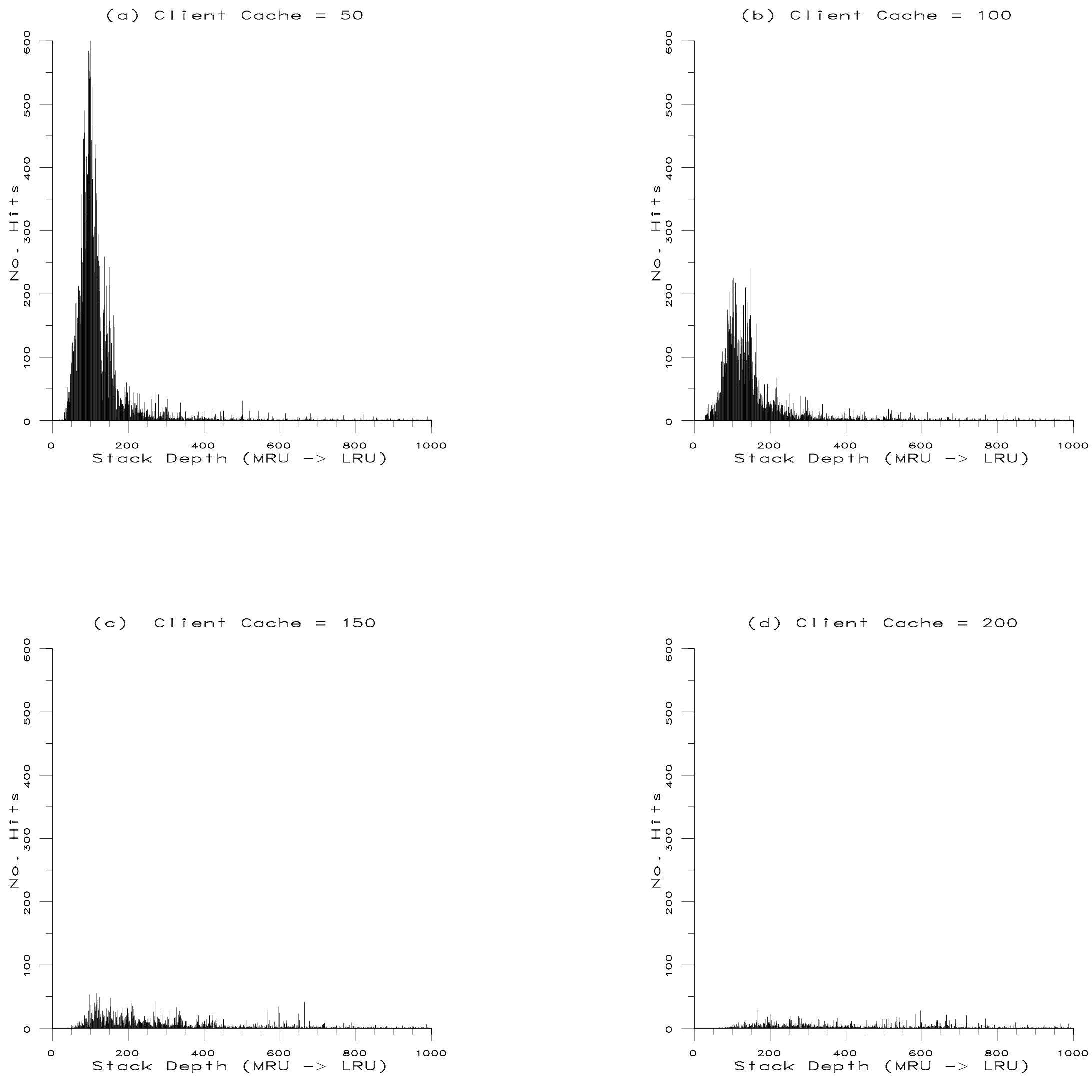

Figure 6: LRU stack depth histograms at an infinite server cache. Each graph is the result of a different client cache size.

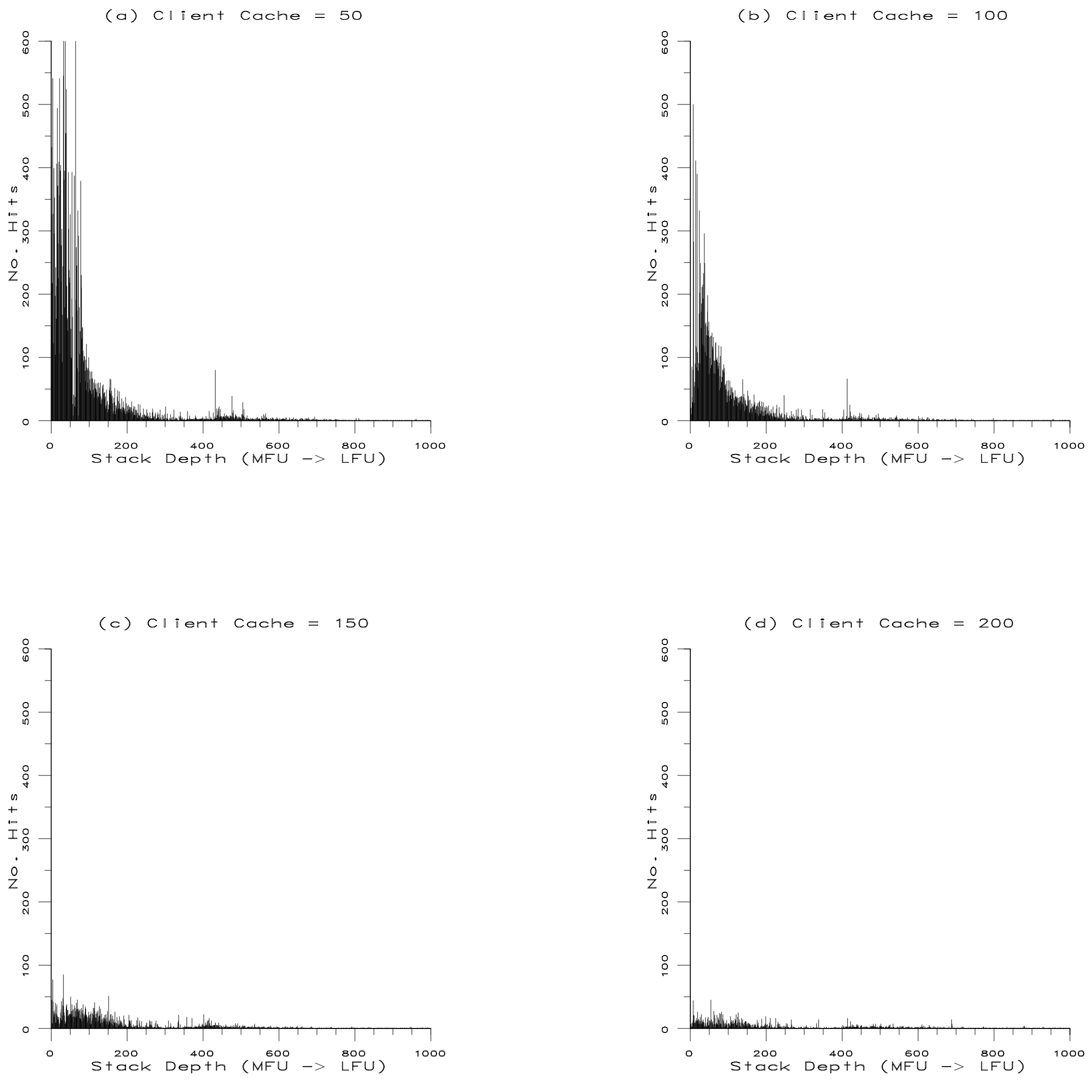

Figure 7: LFU stack depth histograms at an infinite server cache. Each graph is the result of a different client cache size.

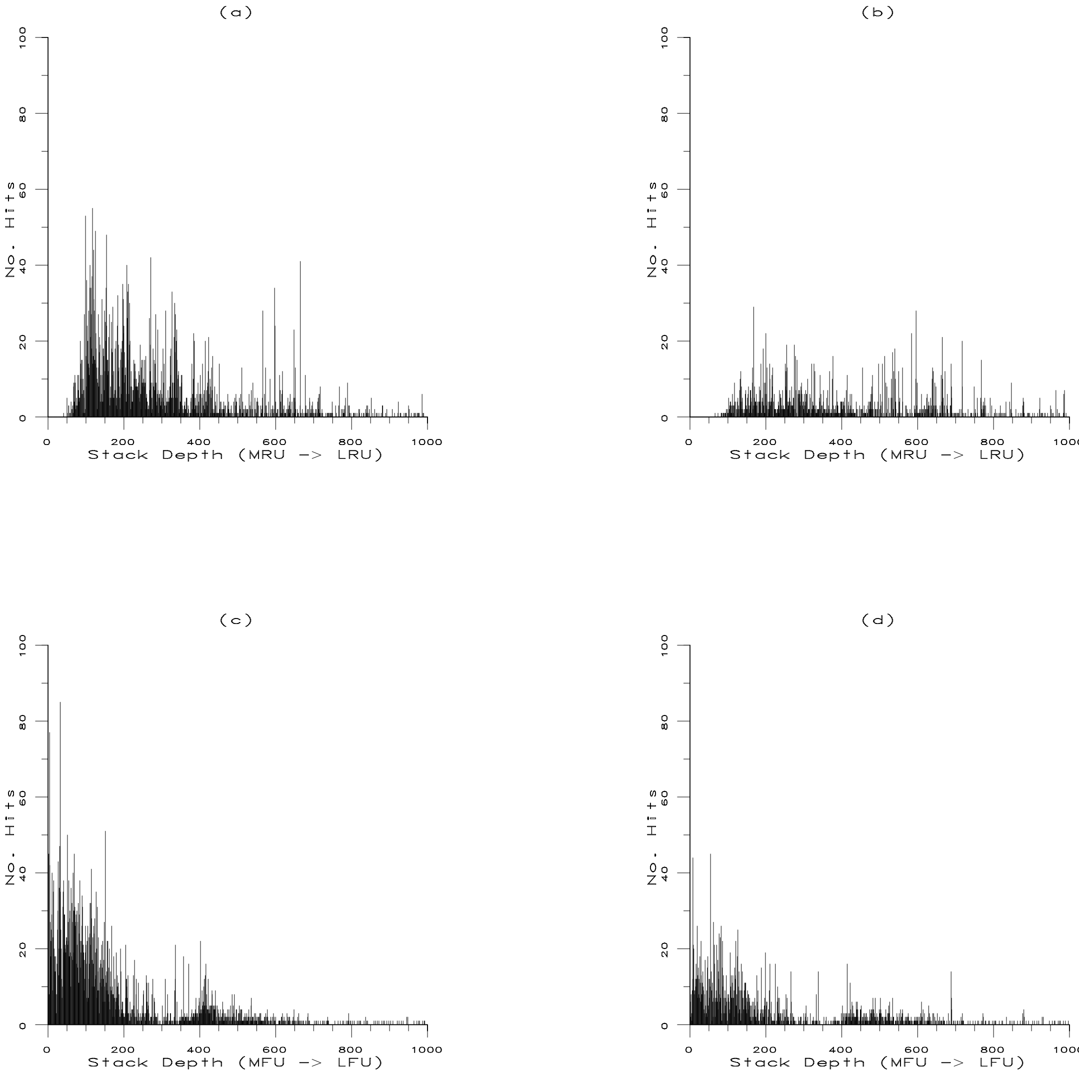

Figure 8: LRU and LFU stack depth histograms for large client cache sizes shown in more detail. Figures (a) through (d) correspond to Figures 6(c), 6(d), 7(c), and 7(d) respectively.

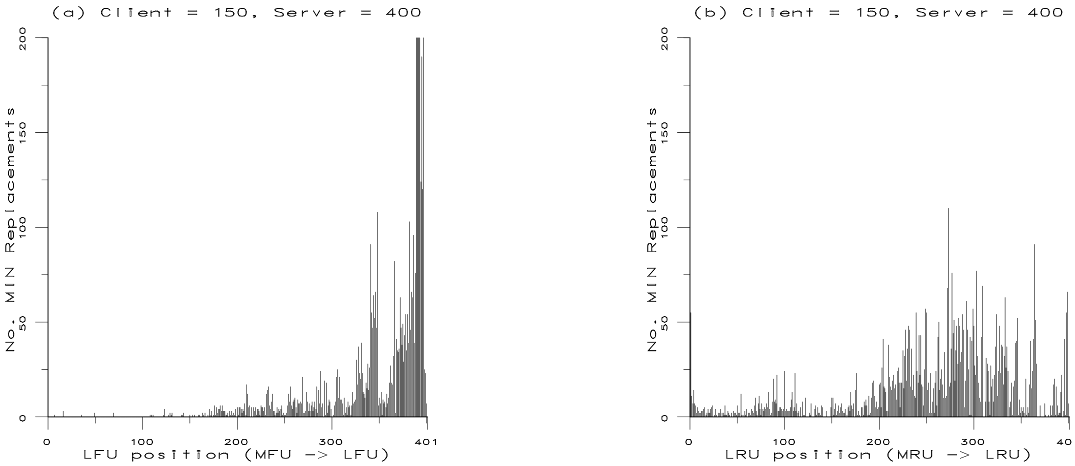

Figure 9: MIN replacement histograms generated at the server cache, for the case where the client cache is 150 blocks and the server cache is 400 blocks.

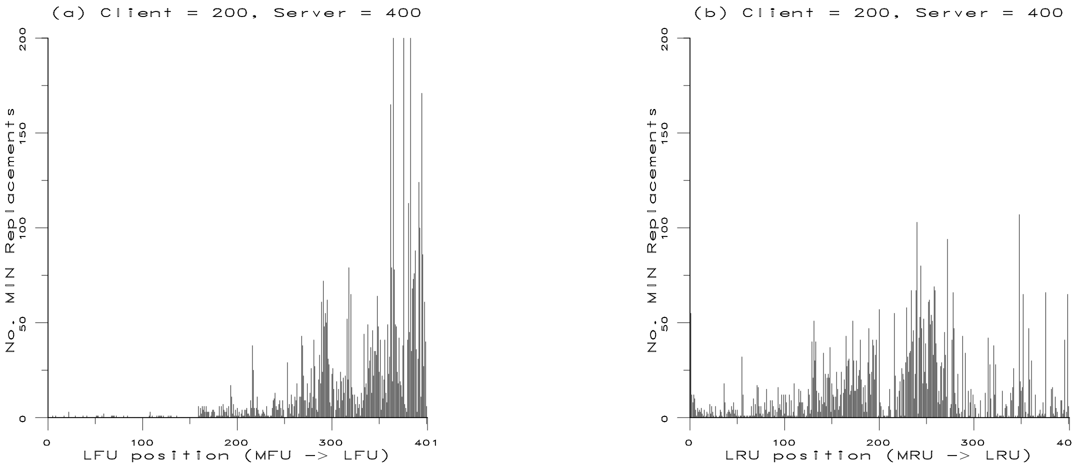

Figure 10: MIN replacement histograms generated at the server cache, for the case where the client cache is 200 blocks and the server cache is 400 blocks.