Title

Analysis of caching algorithms for distributed file systems

Journal

ACM SIGOPS Operating Systems Review, 30(3)

ISSN

0163-5980

Authors

Reed, Benjamin
Long, Darrell DE

Publication Date

1996-07-01

DOI

10.1145/230908.230913

Peer reviewed

Analysis of Caching Algorithms for Distributed File Systems

Benjamin Reed and *Darrell D. E. Long*

Department of Computer Science
University of California
Santa Cruz, CA 95064

Abstract

When picking a cache replacement policy for file systems, LRU (Least Recently Used) *has always been the obvious choice, because of the temporal locality found in programs and data. However, in the case of Sun NFS servers much of the locality is filtered out by the client cache. It has been conjectured that this filtering of locality by the client caches render LRU ineffective as a server cache replacement policy. This study disproves the conjecture by simulating a NFS server cache with real world traces. Traces were taken of NFS read and write requests sent to an NFS server. These traces were then run through a cache simulator using six different cache replacement policies: Least Recently Used* (LRU), *Least Frequently Used* (LFU), *Frequency Based Replacement* (FBR), *First In First Out* (FIFO), *Random* (RAND), *and Optimal* (OPT). *RAND and OPT were used to provide lower and upper bounds on performance. Results show that LRU is an effective NFS server cache replacement policy and frequency based tend to exhibit erratic behavior in the presence of temporal locality and sequentially accessed files.*

1 Introduction

Caching has proven to be a very effective way to reduce the penalty of accessing data on disks. UNIX file access patterns tend to exhibit temporal locality which makes LRU an effective cache replacement policy. When the files reside on an NFS server, the server cache only receives requests that can't be serviced by the client cache. It would seem that LRU performance would go down dramatically if the temporal locality of the requests get filtered out by the client cache. This study seeks to determine whether LRU is a good replacement policy to use in a server.

In order to examine LRU's performance, it is compared with optimal and random replacement policies. In addition, various studies have shown that frequency-based policies perform well with virtual memory [4]. Two versions of frequency-based algorithms are used as comparison.

The study is limited to studying the caching performance of Sun NFS [6] servers, since they are the defacto industry standard. The requests being examined are limited to read and write, since directory information in NFS is opaque it is impossible to simulate the effect of a directory lookup using just the information in the request. One of the most popular directory operations is the lookup operation which contains the directory i-node and the name of the file to lookup; however, in order to simulate properly the offset in the directory of the file would have to be known. In addition, since directory information isn't cached very long in the client, very little locality filtering is done in the client. The two servers being studied are a Sun running SunOS serving files found in /usr/local, which end up being mostly executable files, and an SGI running IRIX which serves home directories, mail spools, and research group directories. The results indicate that LRU is still a good replacement policy on servers.

2 Related Research

This study was constructed similar to a study done previously by Bunt *et al.* [1]. They traced calls to the **getblk** system call on a single NFS client using various workloads, and then ran the trace through a simulator that simulated various sizes of client and server caches. Their results show the frequency-based policies performed significantly better than LRU as the client cache increases.

We decided to redo the experiment, but to avoid modifying the kernel we traced NFS requests as they went across the ethernet. By tracing the network instead of the kernel we were able to trace normal workloads from multiple NFS clients. We chose to simulate the same cache replacement policies as the previous study and to have the cache simulate a Berkeley fast file system [3].

3 Data Collection

One of the advantages to simulating an NFS server cache is that traces are easy to get since the requests are passed over the network. NFS uses the Sun RPC [5] (Remote Procedure Call) protocol to send requests from the NFS client to the server. RPC is implemented on top of UDP [8] (User Datagram Protocol). NFS has reserved UDP port 2049 for exclusive use. The client sends call requests to the server and the server replies with data and/or return codes.

Traces were collected by listening to all the call requests being sent to the NFS file server. This was done by putting the ethernet interface on the same network as the server in promiscuous mode. In promiscuous mode, the ethernet interface will read every packet that goes by on the network. If the computer is on the same network as the file server, it will be able to gather information about all the requests sent to the server. A major advantage of this approach is that it doesn't interfere with the performance of the client or the server, thus eliminating any anomalies that can arise from the monitoring tool affecting the performance of the system.

In order to cut down the number of packets processed, the NIT_PF packet filter was used. NIT_PF is a STREAMS packet filter in SunOS that only lets through packets that meet a certain criteria. The filter was set up to let through only those packets destined for UDP port 2049 of the server and only those that are call requests.

The information needed for the simulator is extracted by parsing the packet headers and body. The information recorded in the trace is the current time, the source and destination of the request, the request type, and the information that corresponds to the request type. For a write request, a file handle and an offset is recorded. For a read request, a file handle, offset, and count are recorded.

Unfortunately, the NFS protocol defines the file handle as being opaque, or basically without meaning to anyone but the server. It also doesn't require that there be only one handle for a file, so in order to use the file handle it must be broken into components. The servers being traced were a Sun running SunOS and an SGI running IRIX. Since the NFS source code wasn't available for either system, the make up of the file handle had to be determined empirically. After repeated observations, it was found that the first two long words represent the device major and minor number, and that bytes 12–15 correspond to the i-node of the file on the server. Given the i-node of a file and an offset we can uniquely identify each block of the file.

4 File system cache simulation

In order to simulate a file system cache, the i-nodes and offsets collected in the data collection phase had to be translated into disk blocks. To make the translation, the file system was assumed to be a Berkeley fast file system [3]. In the Berkeley fast file system, the i-node contains some pointers to the first few blocks of the file, followed by a pointer to a block containing the pointers to the following blocks of the file, and if necessary yet another pointer in the i-node is a double indirect pointer the following blocks. In the simulator, the location of the first 12 blocks are contained in the i-node. The locations of blocks 12 to n, where $n=(block\ size)/4 + 12$, will be located in the indirect block. The location of the indirect block is in the i-node. Blocks greater than n will involve an additional indirect block to find the location.

Each NFS read/write request will generate a block request to fetch the i-node, possibly a request for indirect blocks, and finally a request for the block containing the data to be read. In the simulator a disk block is represented as a tuple of (i-node, indirection, offset / block size). If the tuple is not in the cache and the operation is a read, a miss is recorded. Since writes will be preceded by the client doing a read if necessary, writes are not counted as a cache hit or miss. However, blocks read or written will populate the cache on a miss. The block that is to be removed from the cache is chosen based on the replacement policy.

5 Cache replacement policies

The file system cache simulator simulates six different policies for choosing a block to be replaced. The optimum (OPT) cache replacement policy is used to set an

upper bound on performance. The random (RAND) policy is used has a baseline. A policy is ineffective if it performs worse than RAND. Least recently used (LRU) is a popular replacement policy that replaces the block in the cache that was used least recently. First in first out (FIFO) is a simple queue-like policy that is easy to implement and known to have performance anomalies. Least frequently used (LFU) and frequency based replacement (FBR) were chosen because they have been shown to be effective in previous studies.

5.1 OPT

The optimal replacement policy chooses a block to be replaced based on when it will be used again in the future. The block that will be used farthest in the future is the one chosen to be replaced. It has been shown that no other replacement policy will perform better than this one [2]. While, OPT can't be implemented in practice since that would require the ability to look into the future, it does set an upper bound on performance.

OPT was implemented in the simulator by making two passes of the data: one pass to build a table of future accesses, and another pass to actually simulate the cache using the future access times of the table. This is a very straightforward way of implementing OPT. It can also be implemented in one-pass by using look-ahead; however, for simplicity the two-pass method was chosen.

5.2 RAND

The random replacement policy chooses a block to be replaced by picking one at random. It is very easy to implement. Since the decision is purely random, an effective algorithm should be able to outperform RAND, thus giving us a lower bound on acceptable performance.

5.3 LRU

Least recently used replacement exploits the temporal locality of data. Temporal locality means that if a given block is accessed, it will probably be accessed again sometime soon. Thus, if one wants to approximate OPT assuming temporal locality, the block to be replaced is the one that was accessed the farthest in the past (or least recent). LRU is used in a wide variety of caches and has been shown to be effective in practice.

LRU was implemented in the simulator using a queue. When a block is to be replaced the block at the head of the queue is chosen. Whenever a hit is made by a read or write in the cache, that block is moved to the tail of the queue. The block at the tail of the queue is always the last block accessed and the block at the head is the block that was accessed the farthest in the past.

5.4 FIFO

First in first out replacement is a simple policy that has less overhead than LRU, but doesn't exploit temporal locality as much. The block chosen to be replaced in FIFO is simply the oldest block in the cache. It is implemented in the simulator using a straightforward queue. When a new block is put into the cache, it is added to the tail of the queue. When a block is chosen to be replaced, the block at the head of the queue is chosen. FIFO has been shown to exhibit Belady's anomaly [7] which is manifest as a decrease in cache performance when cache memory is increased.

5.5 Frequency Based

Frequency based algorithms try to find the popular blocks and keep them in cache. The basic frequency-based replacement policy in the simulator is the least frequently used (LFU). In LFU the block that is chosen to be replaced is the block that is used the least. The implementation of the frequency based policies were implemented as described in the study done by Bunt *et al.* [1]. To keep track of the least used block, a count of accesses is kept for each block in the cache. The count is incremented only on read hits. Write hits do not increment the count. When a block is to be replaced, the one with the lowest access count is chosen. Ties are broken by choosing the one that was least recently used.

This implementation of LFU is only approximate since the frequency count is kept only for those blocks in the cache. To prevent some blocks from building up a large frequency count and never being replaced, all the counts are halved when the average frequency count reaches a certain threshold. The thresholds used in this study were the same as the ones used in the previous study.

To implement LFU the blocks of the cache were kept in most-frequently-used order. When a block is to be chosen to be replace, the blocks are examined starting at the most recently used, moving to the least recently used. The last block found with the lowest count is the one to be replaced.

The other frequency-based policy that is in the simulator is the FBR policy. This policy tries to filter out the temporal locality in the data by keeping the blocks of the cache in most-recently-used order and dividing the cache up into three partitions. The first partition is the most recently used part of the cache and occupies *F_new%* of the cache. The last partition is the least recently used part of the cache and occupies *F_old%* of the cache. The partition in the middle are those blocks that aren't in either the old or new partitions. The temporal locality is filtered out by not updating the frequency counts of the blocks in the first partition on a hit. Replacement blocks are chosen from the last partition. The middle partition allows the blocks time to build up frequency counts before being replaced.

6 Experiments

Two different file servers were used in the study. One was a SGI running IRIX. It is a general file server serving home directories, mail, project areas, etc. The other was a Sparc running SunOS that served applications in a /usr/local directory. Both servers are used by the students and faculty of the Computer Science and Engineering Department.

Six traces were obtained from the two systems. Each trace had a duration of two days. The traces were of normal workloads and no artificial loads were introduced. Trace sizes ranged from 100,000 to 400,000 read/write requests.

7 Results

Figure 1, 2, and 3 show the results from three of the traces. LFU 100 and LFU 20 are the least-frequently-used policy with frequencies being halved when the average frequency reaches 100 and 20 respectively. FBR 25:40 is the frequency-based replacement policy with the most recently used partition size of 25% of the total cache size and the least recently used partition size of 40% of the total cache size. FBR 25:60 is the same as FBR 25:40 except that the least recently used partition is 60% of the total cache. The *x*-axis is the size of the server cache in kilobytes and the *y*-axis is the percentage of read resulting in a miss.

In figure 1, the FBR policy exhibits erratic behavior, when the cache size was increased from 1000K to 2000K the percentage of misses went up from 55% to 70%. The jump occurred again for other increases. In contrast LFU seemed to be monotonically decreasing; however, LFU 100 never was able to perform as well as RAND.

Figures 2 and 3 show that the performance of the policies are even more erratic when simulating the SGI file server cache. All of the frequency-based policies performed much worse than RAND with the exception of FBR which occasionally performed better. In theory LFU is a stack algorithm. This means that for a given set of references the set of blocks in a cache will be a subset of the blocks in a larger cache, which implies that an increase in cache size can only result in the number of hits staying the same or increasing. Figures 2 and 3 show that this is not the case for our implementation of LFU. The reason for this is that in theory the frequency count is kept for all the blocks, not just those in the cache. In these frequency-based implementations, when a block leaves the cache its frequency is lost and is reset when it enters the cache.

In all three cases FIFO and LRU perform as expected. They are both consistently better than RAND and don't exhibit the erratic behavior that the frequency-based policies exhibit.

Since frequency-based algorithms generally have trouble with temporal locality, we decided to examine the temporal locality of the data. The LRU simulator was modified to record the depth of a cache hit in the cache. The traces were then run through the LRU simulator with a cache size of 400 megabytes (which in practical terms approximates an infinite sized cache.) Figure 8 shows the raw number of hits occurring at a given depth in the cache. The great majority (over 80%) occur below depth three.

The fact that the hits occur above depth three suggests that they are result of the metadata. If a 16 kilobyte file is read, there will be two requests for 8 kilobyte blocks of the file, since NFS can request a maximum of 8 kilobyte at a time. Both requests will access different data blocks; however, they must both lookup the i-node of the file to find the data block. As files are read in sequentially or in chunks greater than 8 kilobytes the i-node and indirect blocks will be temporally local.

To investigate the effect metadata (i-nodes and indirect blocks) has on the performance of the policies only the data blocks were used in the simulator. Figure 7 shows that the hits in the cache are more uniformly distributed

among the cache depths, nothing like the spike that occurred at depths below 3 in figure 8.

Figure 4 shows that the performance of the frequency-based algorithms have smoothed out; however, note that FBR 25:40 out performs the other policies at a cache size of 4000 kilobytes, but after 6000 kilobytes ends up being the worst performer. In contrast FBR 20:60 consistently performs as well as or better than FIFO and LRU.

Figures 5 and 6 show that even without metadata, the performance of the frequency-based algorithms still tend to be erratic. Interestingly, the performance of the Sun Sparc server seems to indicate that the erratic behavior stopped when the metadata was removed; however, the performance of the SGI indicates that the behavior is still erratic. This is probably due to the different type of files that are being served by the two servers. The Sun is serving /usr/local which has a large number of commonly-used executables. The clients using this server generally do demand paging and use the executable file to page from instead of copying it to a local swap device. The frequency-based policies used have been shown to work well with virtual memory, and in effect by paging from the NFS server the most of the traffic will be virtual memory faults.

The SGI is used for home directories, mail spools, and project directories. These types of files are generally processed sequentially, which could account for the different behavior of the performance of the frequency-based algorithms on the Sun and SGI.

8 Limitations

It should be pointed out that this study has a few limitations. The first being that under a high load, the server may not be able to grab and process packets fast enough. Care was taken to process packets as fast as possible, but the possibility for occasionally missing a packet does exist. Also, because of the indempotent nature of RPC, it is possible that some 2of the requests were actually a duplicate of a previous request, thus giving a false view of the accesses.

A large percentage of NFS requests are requests for directory information. Since this study concentrates on read/write requests, effects of the directory information requests on the server cache are not taken into account.

9 Further Work

Temporal locality and work loads consisting of sequentially accessed files seem to adversely effect the behavior of the frequency-based policies. It would be interesting to specify exactly what the conditions are that make the erratic behavior surface.

It would be interesting to simulate LFU by maintaining a frequency count for all of the blocks in the system. While it wouldn't be practical to implement, it would give insight into the possible performance advantages of frequency-based algorithms.

10 Conclusions

The results in this study are extremely different from the results in the previous study [1], which found LRU's performance to be poor in comparison to the frequency-based algorithms. Part of the difference is the workloads and the number clients that were traced. And another part was the fact that file handles and offsets had to be mapped to physical disk blocks using metadata that was stored in the file system. This mapping was completely ignored in the previous study. This means that an NFS server cache can't be seen purely as a second level of the client, since it must contain information that is irrelevant to the client.

In the two servers examined, LRU proved to be an effective replacement policy whether or not metadata was taken into account. The frequency-based policies were adversely affected by the metadata as well as workloads consisting of sequentially accessed files. When metadata is taken into account LRU is able to exploit the temporal locality of the metadata, and when ignored, empirically it seems that there is still enough temporal locality to justify using LRU.

Acknowledgments

This work has been supported by the Office of Naval Research under grant N00014-92-J-1807.

References

[1] R. B. Bunt, D. L. Willick, D. L. Eager. Disk cache replacement policies for network filservers. In *Proceedings of IEEE International Conference on Distributed Computing Systems – ICDCS '93*, pages 2-11, June 1993.

[2] I. L. Traiger, J. Gecsei, D. R. Slutz. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, (2):78-117, 1970.

[3] J. L. Peterson, J. S. Quarterman, A. Silbershatz. 4.2BSD and 4.3BSD as examples of the UNIX system. *ACM Computing Surveys*, 17(4):379-418, December 1985.

[4] M. V. Devarakonda, J. T. Robinson. Data cache management using frequency-based replacement. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 134-142, May 1990.

[5] Sun Microsystems, Inc. RPC: Remote procedure call protocol specification version 2. RFC 1057, June 1988.

[6] Sun Microsystems, Inc. NFS: Network file system protocol specification version 2. RFC 1094, March 1989.

[7] L. A. Belady, R.A. Nelson, G. S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. Communications of the ACM, 12(6):349-353, June 1969.

[8] J. Postel, User Datagram Protocol, RFC 768, August 1980.

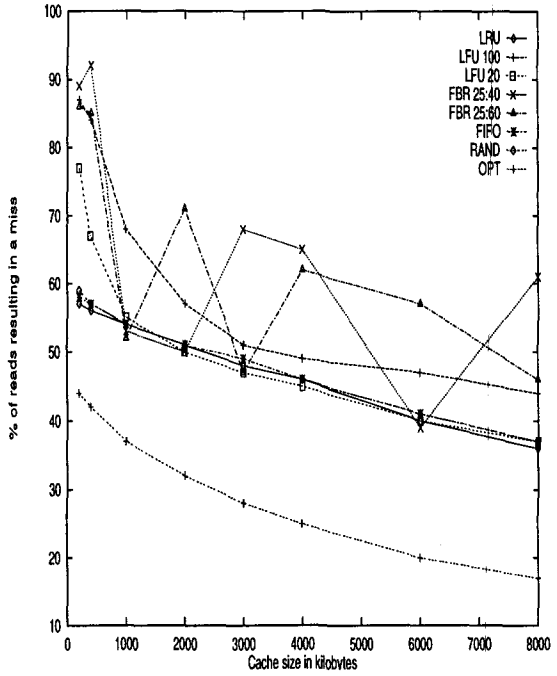

Figure 1: Trace from the /usr/local partition of a Sun Sparc run through the simulator.

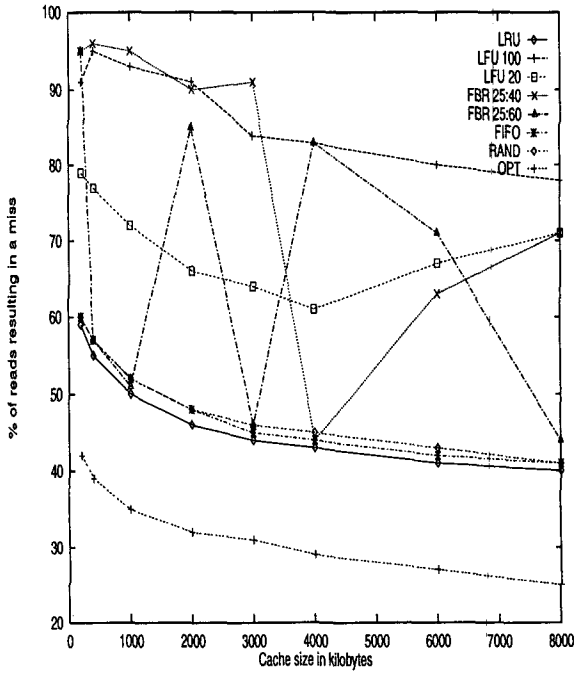

Figure 2: Trace 1 from the SGI run through the simulator.

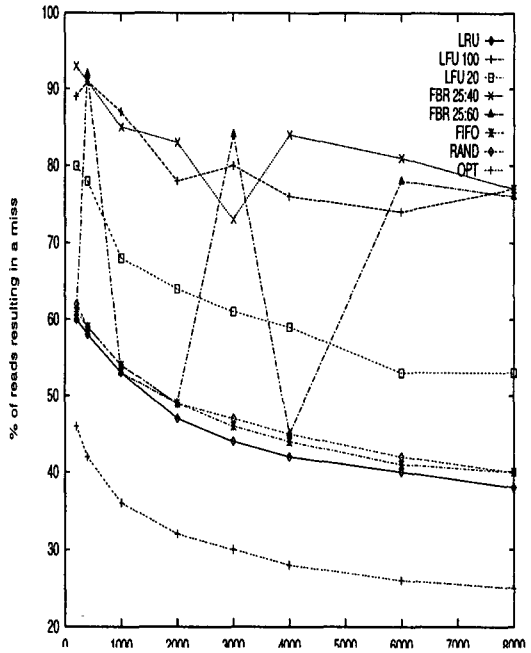

Figure 3: Trace 2 from the SGI run through the simulator.

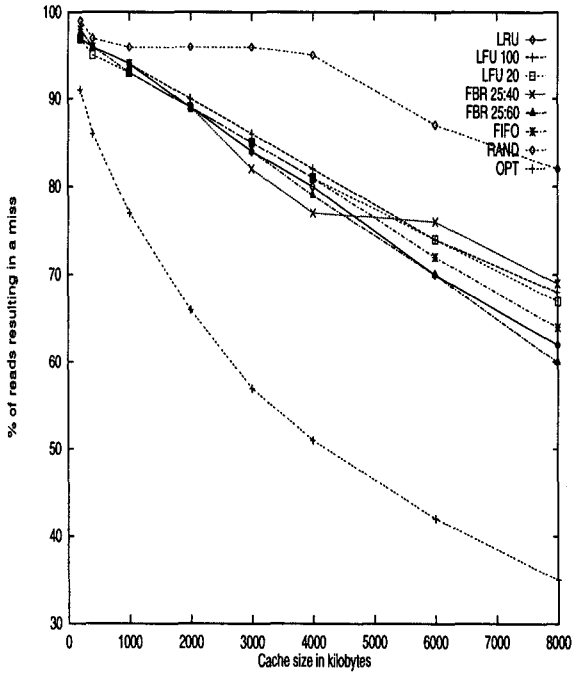

Figure 4: Traces from the /usr/local partition of a Sun Sparc run through the simulator with meta-data ignored.

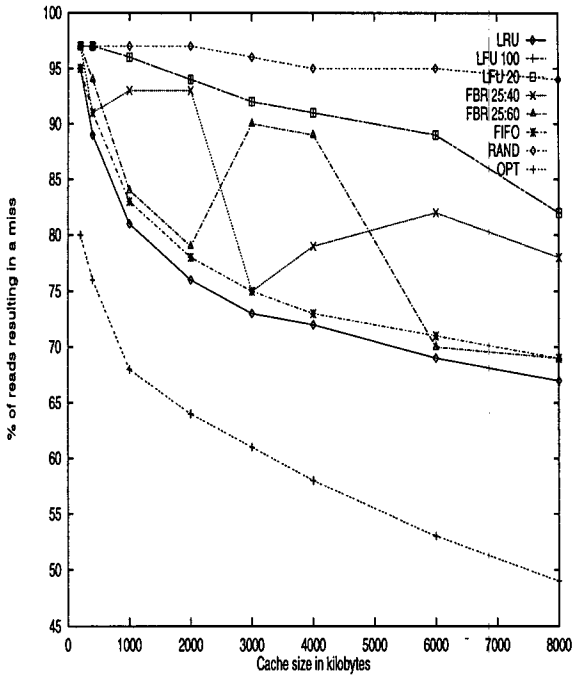

Figure 5: Trace 1 for the SGI run through the simulator with the meta-data ignored.

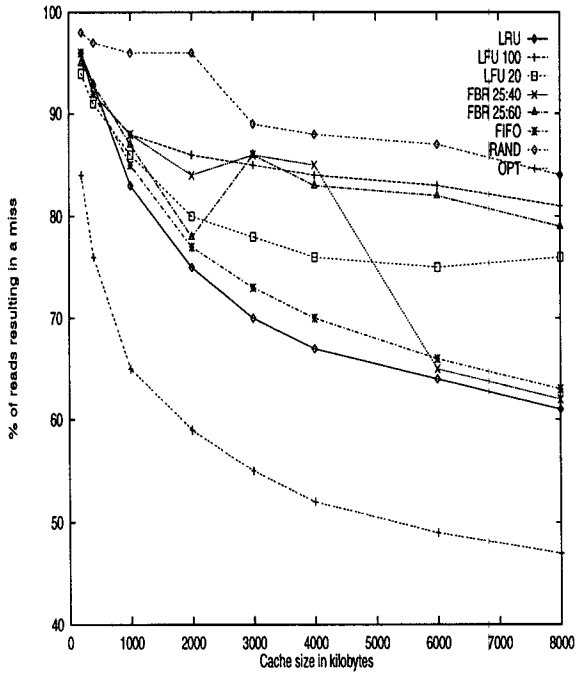

Figure 6: Trace 2 from the SGI run through the simulator with the meta-data ignored.

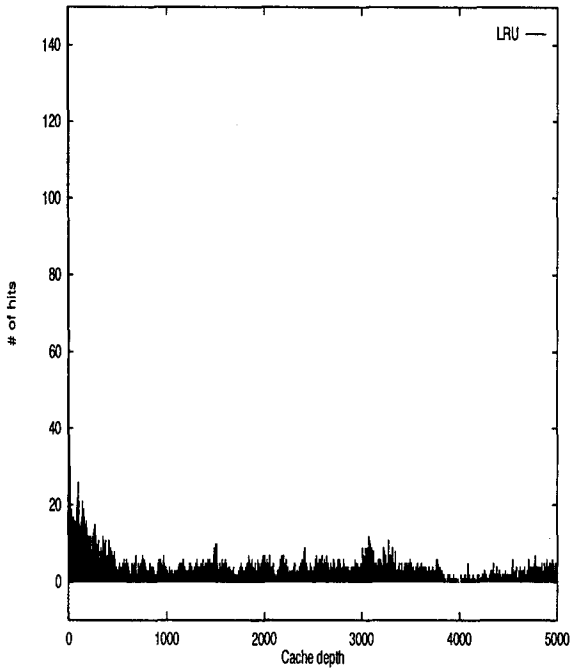

Figure 7: Trace 1 for the SGI run through the infinite LRU simulator counting hits at a given depth and ignoring meta-data

Figure 8: Trace 1 for the SGI run through the infinite LRU simulator counting hits at a given depth