Data Cache Management Using Frequency-Based Replacement

John T. Robinson Murthy V. Devarakonda

IBM Research Division, T. J. Watson Research Center
P. O. Box 704, Yorktown Heights, NY 10598

Abstract. We propose a new frequency-based replacement algorithm for managing caches used for disk blocks by a file system, database management system, or disk control unit, which we refer to here as data caches. Previously, LRU replacement has usually been used for such caches. We describe a replacement algorithm based on the concept of maintaining reference counts in which locality has been "factored out". In this algorithm replacement choices are made using a combination of reference frequency and block age. Simulation results based on traces of file system and I/O activity from actual systems show that this algorithm can offer up to 34% performance improvement over LRU replacement, where the improvement is expressed as the fraction of the performance gain achieved between LRU replacement and the theoretically optimal policy in which the reference string must be known in advance. Furthermore, the implementation complexity and efficiency of this algorithm is comparable to one using LRU replacement.

1. Introduction

In any memory hierarchy performance can be improved by caching data from a lower level of the hierarchy in a higher level. The goal in designing a cache management algorithm is to achieve as low a cache miss ratio as possible subject to constraints on the complexity of the algorithm. Cache management algorithms have been most widely studied in the context of CPU caches (for example see [SMIT82]), in which there are significant complexity constraints. There have also been numerous studies concerning virtual memory management, in which the complexity constraints are somewhat relaxed (a hit must be handled quickly, e.g. by means of address translation and the setting of a reference bit, whereas a miss can be handled by operating system software). Here, however, we are primarily concerned with caches residing in main memory that are managed entirely by software. Examples include caches for file systems and database management systems. In these cases the average access time for file or database disk blocks is reduced by storing some subset of these blocks in the cache. Another example is a cache within a disk control unit (disk cache), managed by control unit microcode, which can be used to reduce the average device access time without involving host system software. Here we will refer to these various types of caches as *data caches*.

The design issues for data caches are somewhat different than those for CPU caches or virtual memory. For example, due to the fact that the cache is managed by software (or control unit microcode), a strict implementation of LRU replacement is possible. In addition to relaxed complexity constraints, the reference behavior may be different.

Previous studies of file system caches and disk caches include [OUST85] and [SMIT85], respectively. Issues of block size, cache size, and so on were investigated, but in both studies an LRU replacement policy was assumed throughout. On the other hand, various replacement policies were studied for database caches in [EFFE84]. Due to factors related to a number of other design issues, this study did not conclusively show that any one replacement policy was best. However, it was concluded that LRU replacement and the CLOCK algorithm (which approximates LRU replacement using reference bits, and has been used in virtual memory management) gave good satisfactory overall behavior.

In terms of differences in reference behavior, in general it seems that there is less locality (the degree to which recently referenced data tends to be re-referenced) in data caches as compared to CPU caches or virtual memory systems (this was one of the conclusions of [EFFE84] for database caches). If there were in fact a small degree of dependence of each reference on previous references, then it is possible that an independent reference assumption (in which each block is referenced with a given fixed probability) could be used to accurately model reference behavior. In such a case the optimal cache management policy would be to keep the blocks with the highest reference probabilities in the cache at all times (see, e.g., [COFF73]). In practice of course there would be a problem in determining these probabilities. However, given a reference string produced by a trace, the blocks can be sorted by reference frequency, and this policy can be simulated. It has been our experience

that for the file system traces we describe later this leads to poor performance as compared to LRU replacement. The conclusion is that although there is less locality than in CPU caches or virtual memory, there is still sufficient locality so that LRU replacement performs quite well. Similarly, in [EFFE84] replacement algorithms based on estimating reference probabilities did not consistently outperform those using LRU replacement. This agrees with general experience: currently, LRU replacement is almost universally used in software managed buffers or caches.

Here, we study a new frequency-based replacement cache management algorithm based on the concept of "factoring out locality" from reference counts. Like previously studied algorithms, reference counts are maintained for each block in the cache; unlike previous algorithms, reference counts are not incremented on every reference. Instead, reference counts are incremented only for those blocks that have aged out of a "new section" of the cache. Another novel aspect of the algorithm is that replacement choices are confined to those blocks that have aged into an "old section" of the cache. These techniques and the intuition behind their use are discussed in Section 2, in which we describe the frequency-based replacement cache management algorithm.

In Section 3 we present performance analysis results for the new algorithm. These results were generated via detailed trace-driven simulations for five actual systems, with different hardware configurations, operating systems, and/or workload characteristics. The performance of the new algorithm is compared to LRU replacement and the theoretically optimal replacement algorithm in which the reference string must be known in advance. The results indicate that significant performance improvements can be obtained for most systems at small to intermediate cache sizes, and for all systems at intermediate cache sizes (for increasingly large cache sizes, the performance of all three replacement algorithms tends to converge). Finally, in Section 4 we discuss the major conclusions of our study, and identify some areas for further research.

2. Frequency-Based Replacement

In this section we present the frequency-based replacement (FBR) algorithm by means of a sequence of refinements. We discuss (1) the use of a "new section" which controls the incrementing of reference counts; (2) the use of an "old section" to confine replacement choices to blocks that have not been too recently referenced; and (3) periodically adjusting reference counts so as to approximate reference frequencies and avoid practical problems. Finally we describe an implementation which is comparable in complexity to an implementation of a cache management algorithm using LRU replacement.

2.1. Factoring Out Locality

The basis of our cache management algorithm is the LRU replacement algorithm, in which logically the cache consists of a stack of blocks, with the most recently referenced block always pushed on the top of the stack. Unlike

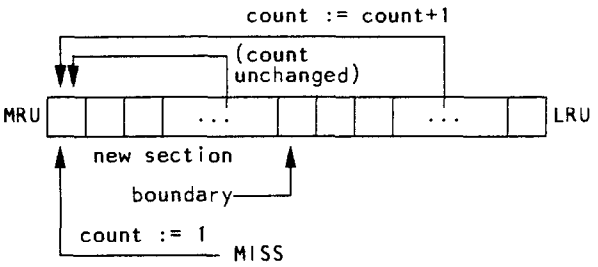

Figure 2.1. Use of the New Section

LRU replacement, the least recently used block will not necessarily be selected for replacement on a cache miss. Instead, a reference count is maintained for each block in the cache, and in general the blocks with the smallest reference counts will be candidates for replacement.

When a block is first brought into the cache, its reference count is intialized to one. Previous algorithms using reference counts have incremented the count for a given block on every reference to that block. This results in the following kind of problem: certain blocks are relatively infrequently referenced overall, and yet when they are referenced, due to locality there are short intervals of repeated re-references, thus building up high reference counts. After such an interval is over, the high reference count is misleading: it is due to locality, and cannot be used to estimate the probability that such a block will be re-referenced following the end of this interval (this description is intuitive, since in practice such intervals are not as well-defined as the preceding discussion may suggest).

Here, this problem is addressed by "factoring out locality" from reference counts, as follows. A certain portion of the top part of the stack is set aside as a *new section*. When there is a cache hit, if the block is in the new section then its reference count is *not* incremented, but it is incremented otherwise. This is illustrated in Figure 2.1. Given a sufficiently large new section, this results in the reference counts for blocks that are repeatedly re-referenced within a short interval remaining unchanged. The fraction of blocks in the new section, F_{new}, is a parameter of the algorithm.

2.2. Confining Replacement

The technique just described for factoring out locality from reference counts can be used without further refinement as the basis of a replacement policy: on a miss, select the block with the smallest reference count that is not in the new section for replacement, choosing the least recently used such block if there is more than one.

Preliminary experiments with the policy just described showed that although it was sometimes possible to obtain slightly better performance than LRU replacement, consistently significant improvements could not be obtained, and

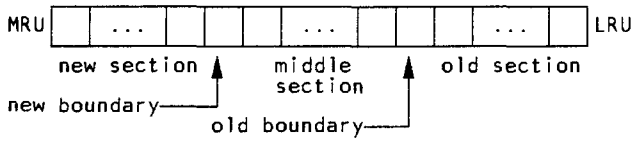

Figure 2.2. Three Sections of the Cache

often the performance was worse. The problem seemed to be the following: (1) on a cache miss, a new block is brought into the new section and initialized with a count of one; (2) the count remains at one as long as the block remains in the new section; (3) eventually the block ages out of the new section, with its count still at one; (4) if the block is not now re-referenced fairly quickly it is very likely to be replaced since it necessarily has the smallest reference count of those blocks that are not in the new section, and all other less recently used blocks with a count of one may have already been replaced. In other words, there did not seem to be a sufficiently long interval for blocks aging out of the new section to build up their reference counts even if they were relatively frequently referenced.

This problem is addressed by confining replacement, as follows. A portion of the bottom part of the stack is defined as the *old section*, and we call the remaining part of the stack between the new and old sections the *middle section*. This is illustrated in Figure 2.2. Using the old section, the replacement policy is as follows: on a cache miss, select the block with the smallest reference count in the old section for replacement, choosing the least recently used such block if there is more than one. Assuming a sufficiently large middle section, intuitively this allows relatively frequently referenced blocks a chance to build up their reference counts before becoming eligible for replacement. The fraction of blocks in the old section, F_{old}, is another parameter of the algorithm, where we require that $F_{old} \leq 1 - F_{new}$. Note that (1) by setting $F_{old} = 1 - F_{new}$, the middle section is empty, and we have the replacement policy described at the beginning of this section; and (2) by setting $F_{old} = 1/(\text{cache size})$, the old section consists of one block, and we have the LRU replacement policy.

2.3. Adjusting Reference Counts

There are some practical problems associated with the use of reference counts. One is overflow, which is easily dealt with by not incrementing a count if it is currently at a given maximum value. Another problem, not quite so easily handled, is that even using the new section as described above, certain blocks may build up high reference counts and never be replaced: such blocks are essentially fixed in the cache. Now it could be that such blocks *should* be fixed in the cache since they are among the most frequently referenced blocks. However, in the absence of additional information, it must be assumed that it is equally likely that the high reference frequency for such a block is short-lived. In this case fixing such blocks in the cache would cause a gradual degradation in performance. Another issue is relating reference counts to reference frequency (that is, references per unit of time, where time could be defined as total references).

One approach to these issues was described in [EFFE84], where the concept of *reference density* was introduced. There, the reference density of a block was defined as the reference count for the block divided by the total number of references made since the block was brought into the cache. Unfortunately reference density, defined in this fashion, is a floating point number, and given a set of blocks, there is no simple fast way to find the block with the minimum reference density (at least compared to the implementation we discuss below, in which the least recently used block with the smallest reference count in the old section is almost always found directly). For this reason we follow another approach mentioned in [EFFE84], which was classified there as "periodic aging by division".

More precisely, we dynamically maintain the sum of all reference counts. Whenever the average reference count exceeds a predetermined maximum value, $A_{\max}$ (another parameter of the algorithm), every reference count C is reduced to $\lceil C/2 \rceil$. Thus, in the steady state the sum of all reference counts stays between $N \times A_{\max}/2$ and $N \times A_{\max}$ (where N is the number of cache blocks), and the reference count of a block is approximately linearly related to the frequency of references to the block while it is in the middle and old sections. Note that in this reduction of reference counts, a count of one remains at one, a count of two is reduced to one, a count of three is reduced to two, etc. As will later be seen, most blocks selected for replacement have a reference count of one, and it is desirable for efficiency to maintain this property in the steady state. Therefore, after reference counts have been reduced, we want to have a count of two reduced to one (since using division with a fixed divisor and integer arithmetic, the only other alternative is to have it remain at two indefinitely). For this reason we did not make the divisor a parameter above (as it was in [EFFE84]), since division by two using the ceiling function yields the desired behavior. Finally, we did not consider reduction of reference counts by subtraction, since in that case there is no intuitive relationship between reference counts and reference frequencies.

2.4. Implementation

The preceding discussions essentially define the frequency-based replacement algorithm at a logical level; in this section we discuss an efficient implementation. As a starting point we assume the following typical implementation of a cache management algorithm using LRU replacement: (1) the cache directory consists of a collection of cache directory entries, one for each block in the cache; (2) each cache directory entry (CDE) describes the contents of the associated cache block (e.g. file name and block number) along with any additional information associated with that block; (3) a CDE for a given block is searched for using a hash table, where CDEs with the same hash value are linked together in a "synonym" chain to handle collisions (for example); (4)

the LRU stack is implemented as a doubly-linked list of CDEs, called the *LRU chain*, and there are pointers to the head (most recently used block) and tail (least recently used block) of the list; (5) on a cache hit the CDE for the block is moved to the head of the list, and (6) on a cache miss the block associated with the CDE at the tail of the list is selected to be replaced, this CDE is set to refer to the new block, and is moved to the head of the list. Since there is a one-to-one correspondence between the CDEs and the blocks in the cache, we will sometimes use the term "block" for CDE without ambiguity.

First we describe how the three cache sections are implemented. In addition to the pointers to the head and tail of the LRU chain, pointers to the most recently used blocks in the middle and old sections, respectively called the *new boundary* and the *old boundary*, are maintained. These pointers were indicated earlier in Figure 2.2. Also, each CDE contains two boolean fields: (1) a *new* field, which is true if the block is in the new section and false otherwise, and (2) an *old* field, which is true if the block is in the old section and false otherwise. As blocks are moved in the LRU chain the boundary pointers are adjusted so as to keep the sizes of each section constant, and the *new* and *old* fields of CDEs are set appropriately. For example, suppose there were a hit on a block in the old section. This block will be moved out of the old section to the head of the LRU chain. Therefore, in this case: (1) *new boundary* is set to point to the next more recently used block in the LRU chain, and the *new* field for this block is set false; (2) *old boundary* is set to point to the next more recently used block in the LRU chain, and the *old* field for this block is set true; and last (3) the block that was referenced is moved to the head of LRU chain, with *new* set true and *old* set false.

Next we describe how the least recently used block with the smallest reference count in the old section is found efficiently. Each CDE contains a *count* field used for the reference count, and two additional pointer fields that are used when the CDE is in a *count chain*, which we now define. For each reference count value 1, 2, 3, ..., C_{max} (where C_{max} is a parameter), a count chain is maintained for that value. Count chain i is a doubly-linked list of all blocks with a reference count of i in the same order that they appear in the LRU chain. Pointers to the head and tail of each count chain are maintained. The algorithm for finding a block to replace is as follows.

1. Do the following for $i = 1$ to C_{max}:
   if count chain i is not empty then
   if *old* is true for the least recently used block in this chain then
   select this block to replace and exit.
2. If no block was found in step (1), select the least recently used block to replace.

Note that if the smallest reference count for blocks in the old section is greater than C_{max}, the least recently used block is replaced by default. We will later see that even with fairly small values for C_{max} this happens very rarely. In fact most of the time (typically more than 90%) a block with a reference count of one is replaced, and of the remaining cases it is most likely for a block with a reference count of two to be replaced, etc. Thus the above search algorithm gives extremely good expected performance, since the expected number of blocks examined is very close to one.

Although there are multiple count chains, each block is in at most one such chain, and therefore the overhead for maintaining the count chains is on the average less than that of maintaining the LRU chain. Maintenance of the count chains can be summarized by the following rules: (1) on a miss, the block being replaced is removed from its count chain (if it is in one), and the new block is put at the head of count chain 1; (2) on a hit to the new section, the block is moved to the head of its count chain (if it is in one); and (3) on a hit to a block that is not in the new section with reference count C, the block is removed from its count chain (if it is in one), C is incremented, then if $C \leq C_{max}$ the block is put at the head of count chain C.

Given the discussion in this and the preceding sections, implementation of the frequency-based replacement algorithm using the above techniques is straightforward. This concludes our description of an efficient implementation.

3. Performance Analysis

This section describes the trace-driven simulation of data caches, and the results obtained from our analysis. First we describe the traces that were used.

3.1. Description of the Traces

The traces used in the simulations were obtained from actual systems running three different operating systems with various workloads. We used a total of five different traces, and each of them was obtained from a system running a workload normal to its environment. The five traces are summarized in Table 3.1.

The first two traces, HAW and KNG, are from the VM HPO operating system running on IBM 3081 processor complexes. The HAW trace was collected at the IBM Watson Research Center and the KNG trace was collected from an IBM development lab. The system at the Watson Research Center was used primarily for research programming, electronic mail, and text processing. The system at the development lab was used mostly for software development work and electronic mail. The data for these two systems was collected on an individual user basis. About 10% of users were randomly chosen from the list of user sign-ons of the system, and for each selected user their logical file usage was traced during their "normal" work day. These traces are described in more detail in [BOZM89], and using techniques described there, the traces were reduced to a single trace consisting of a global string of references to 4K-byte file system blocks.

Trace Name	System Description				Trace Details	
	OS	Hard-ware	Lo-cation	Work-load	Refs	Time
HAW	VM HPO	IBM 3081	IBM TJW Re-search	Re-search Com-puting	88K	(21 user-days)
KNG	VM HPO	IBM 3081	IBM Dev Labs	Prog. and Mail service	87K	(45 user-days)
UNIX	4.3BSD UNIX	VAX 11/780	Univ. of Illi-nois	Re-search Com-puting	2,169K	4 days
C51	MVS systems	3 IBM 3084s	Com-mercial Estab-lishment	Large Business Applica-tions	812K	15 min.
CUST11	MVS systems	1 IBM 3081, 1 IBM 3033	Com-mercial Estab-lishment	Large Business Applica-tions	417K	15 min.

Table 3.1. Summary of Trace Descriptions

The UNIX[1] trace is from a multi-user 4.3BSD UNIX system running on a VAX[2] 11/780 at the University of Illinois. The system was used for text editing, research programming, electronic mail, and bulletin board services by the faculty and graduate students of the Department of Computer Sciences. This workload is similar to that of the HAW trace, and can be characterized as interactive research computing. The trace was collected for four days continuously during a work week using kernel instrumentation. More details of this trace can be found in [DEVA88, DEVA90].

The last two traces, C51 and CUST11, were taken for 15 minute intervals from two large data processing complexes at commercial establishments using MVS operating systems. These systems were used mostly for large business applications, including transaction processing and database queries. The workload on these systems is, therefore, substantially different from that of the UNIX and VM systems. The first three traces -- HAW, KNG, and UNIX -- have enough information to distinguish between read and write accesses. For the MVS traces, we assumed that all references were reads. However, our experiments with the first three traces show that the lack of read/write distinction for the MVS traces may have introduced only a small error. This may be due to the fact that reads usually dominate file access.

[1] UNIX is a registered trademark of AT&T Bell Laboratories.
[2] VAX is a registered trademark of DEC.

The longest of the five traces is the UNIX trace with over 2 million references to 4K-byte file system blocks. MVS traces are of moderate length with about 0.8 and 0.4 million references. VM traces are smaller than the MVS traces.

Each trace record of the reduced VM traces consists of a unique disk block number, and a flag indicating whether the access is a read or write. These represent the file system I/O requests generated by the users for which trace information was collected. The MVS trace records are similar to the reduced VM trace records except that the access type information was not recorded.

The UNIX trace was obtained through the tracing of file-related *system calls* in a way similar to that described in [OUST85], and is fully described in [DEVA88]. Each record of the UNIX trace indicates an accessed area of a file in an open-close sequence, and whether the access was a write. Since an area smaller than a cache-block may have been written into a previously existing file block, simulation of such a trace includes block updates as well as block reads and block writes. A block update is handled as a read followed by write. Further, the UNIX trace also contains file deletion information, and it was used for purging cache blocks. The trace, however, does not include the inode access or directory usage that might occur during a pathname to inode translation. *Execve* system calls were traced, which show what programs were executed and the size of the executable code files. This information was used to simulate program loading activity, assuming that the loading of a program involves reading of the entire program file.

3.2. Cache Simulation

A program was written to simulate data caches consisting of fixed-size blocks. The simulator reads a trace file consisting of trace records, and simulates the actions of a data cache according to the trace record.

For a block read, the data cache is first searched for the block, and if found, there is a cache hit. Otherwise, there is a cache miss, and the following sequence of actions are simulated: the block is fetched from disk (i.e. a *block in*), an unused cache block is found, and the block fetched from the disk is stored in the cache block. If an unused cache block is unavailable, a block in use is freed according to the replacement algorithm, which may involve a write to disk for the contents of the cache block (i.e. a *block out*).

For a block write, the data cache is searched as before, and if found the cache block is marked as modified making it eligible for a block out. If the block is not in the cache, a cache block is made available as in the case of block read. The modified blocks are written back to disk according to the *write-policy* . Various write-policies are known, such as delayed-write, write-through, and intermediate policies such as a 30-second update (see, e.g., [OUST85], [DEVA90]). Although we experimented with various of these policies,

here for simplicity we will present our results only for the delayed-write case, in which a block is written back to disk at the time it is replaced in the cache. We note however that performance improvements similar to the ones shown here are obtained for the other write policies.

3.3. Simulated Replacement Algorithms

Besides the FBR algorithm, two other replacement algorithms, namely LRU and OPT, are simulated for comparison purposes. The LRU algorithm is a straightforward least recently used replacement scheme, where blocks are maintained in the least recently used order, and when a replacement is needed the least recently used block is chosen. As discussed earlier this is a limiting case of the FBR algorithm in which F_{old} is chosen so that the old section contains one block.

The OPT algorithm is the theoretcially optimal replacement algorithm in which the entire reference string must be known in advance. This algorithm is equivalent to that of Belady's optimal algorithm (e.g. see [COFF73]) for page replacement. For a given cache size, the block miss ratio (defined below) for any realizable algorithm in which references are not known in advance must be greater than that obtained using the OPT algorithm.

The performance of the FBR algorithm is presented as it relates to the performance of these two algorithms. Simulation of the FBR algorithm involves use of specific values for its parameters. In general, for a given cache size and workload, parameters that minimize the miss ratio for that cache size can be found by trial and error. However, we found the FBR algorithm to be quite robust: small changes in the parameters yield only small changes in performance. The optimal parameters for a given cache size, though, may not be optimal for a much larger or much smaller cache size.

Here, in order to simplify the presentation of our performance results, we fixed the parameters for each workload over all cache sizes. The parameters we used for each workload are shown in Table 3.2. These parameters were chosen so as to give good overall behavior over a wide range of cache sizes. Note that given any particular cache size and workload, increased performance could be achieved by "fine-tuning" the parameters for that particular case. For example, due to fixing the parameters, for some cases of small cache sizes (as we see below), the performance of FBR was slightly worse than that of LRU for the HAW and KNG traces. However we know that by appropiately choosing parameters FBR can perform at least as well as LRU since it can be made equivalent to LRU. Similarly in other cases in which FBR significantly outperformed LRU even more performance improvement could have been obtained by optimizing the parameters for that particular cache size. It is our experience, though, that in general such optimization would yield only a small improvement over the results shown here due to the previously discussed robustness of the algorithm.

Parameter	Values				
	UNIX	C51	CUST11	HAW	KNG
C_{max}	8	3	3	3	3
F_{new}	25%	25%	25%	25%	25%
F_{old}	60%	20%	40%	20%	20%
A_{max}	100	100	100	100	100

Table 3.2. Parameters Used for FBR Algorithm

3.4. Measures Studied

Block miss ratio, as defined below, is the key performance measure used in this study.

$$Block\ Miss\ Ratio = \frac{Block\ Ins + Block\ Outs}{Block\ Reads + Block\ Writes}$$

In order to show the performance improvement offered by FBR over LRU, we use a measure called *relative improvement.*

$$Rel.\ Improvement = \frac{LRU\ Miss\ Ratio - FBR\ Miss\ Ratio}{LRU\ Miss\ Ratio - OPT\ Miss\ Ratio}$$

Relative improvement shows the improvement offered by FBR over LRU as a fraction of the difference between LRU and OPT (the optimal replacement algorithm). If FBR is as good as OPT, relative improvement will be 1.0 and if FBR is worse than LRU the relative improvement is negative.

Finally, we define a rereference measure for each reference count, which we use later in examining performance characteristics of FBR.

$$Reref(i) = \frac{Number\ of\ rereferences\ to\ blocks\ with\ count\ i}{Number\ of\ references\ resulting\ in\ count\ of\ i}$$

Note that this definition of $Reref(i)$ is dependent on a given fixed position of the new boundary. Given a particular new boundary position, $Reref(i)$ is the probability that an arbitrary block with a count of i is rereferenced.

3.5. Simulation Results

For each replacement algorithm, simulation results were obtained for cache sizes ranging from 800K bytes to 32M bytes. As discussed above, results were obtained for different write-back policies (for VM and UNIX traces), and also for different cache block sizes (for the UNIX trace). Here though we present our results only for the delayed-write policy and for a block size of 4K.

Miss ratios for UNIX, CUST11, and HAW traces are shown in Figures 3.1 through 3.3. Each figure shows miss ratios for LRU, FBR, and OPT replacement algorithms for cache sizes ranging from 800K bytes through 32M bytes. These figures provide examples indicating the general shapes of the miss ratio curves, but in Table 3.3 relative improve-

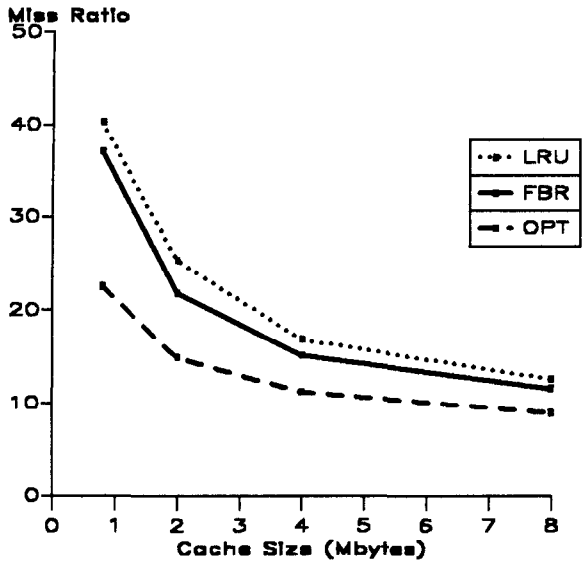

Figure 3.1. Block Miss Ratios (%) for UNIX Trace

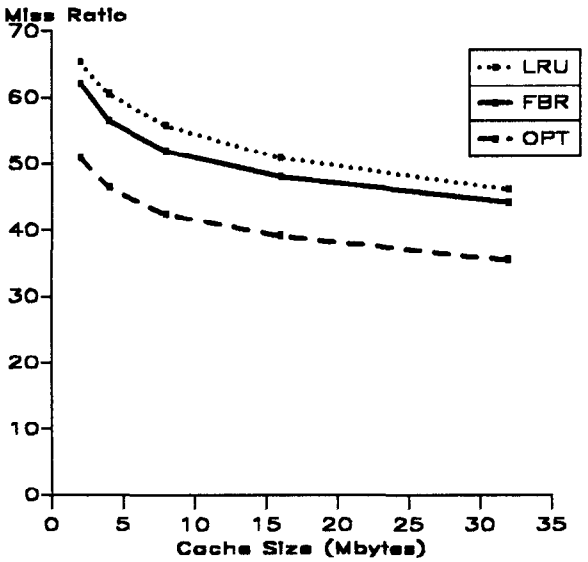

Figure 3.2. Block Miss Ratios (%) for CUST11 Trace

ment (as defined above) is shown for all traces for the same range of cache sizes as in the figures.

From Figure 3.1 and Table 3.3, it can be seen that FBR performs consistently better than LRU for the UNIX trace. Its best performance is at the knee of the miss ratio curve (around 2M or 4M bytes cache size). At 2M bytes cache size, for instance, the miss ratio for FBR is 21.8% whereas the miss ratio for LRU is 25.3%. The relative improvement at this cache size is approximately 34%. At this point the FBR miss ratio is 3.5% less than the LRU miss ratio, which is a decrease of 14%. This implies that in an actual system the file system I/O rate could be reduced 14% by using FBR instead of LRU. As the cache size is reduced or increased from the knee of the curve, the relative improvement is seen

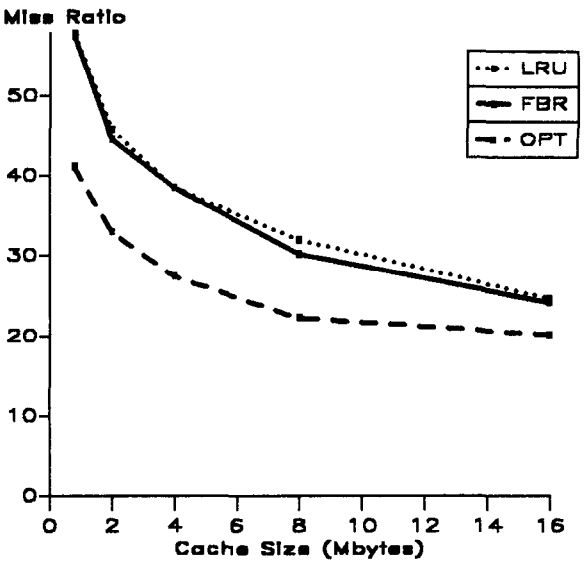

Figure 3.3. Block Miss Ratios (%) for HAW Trace

Cache Size	Percent Relative Improvement				
	UNIX	C51	CUST11	HAW	KNG
800K	17.96			2.16	0.60
2M	33.69	4.63	22.36	9.49	-3.40
4M	30.27	9.21	28.68	-0.09	4.92
8M	26.80	15.88	29.41	18.46	27.42
16M	14.08	15.00	23.61	9.64	14.61
32M		10.46	18.47		

Table 3.3. Rel. Improvement (%) for Various Traces

to fall off. Thus FBR is seen to offer the best performance improvement at the knee of the miss ratio curve, where the cache size is large enough to permit significant optimization and yet small enough to require a proper utilization of available space.

The results for the MVS traces show a similar pattern. For the CUST11 trace FBR is once again seen to consistently outperform LRU at all cache sizes, but its best performance is once again near the knee of the curve, which is at about 8M bytes. At this point a relative improvement of about 29% is achieved.

However, for the VM traces the relative performance of FBR is somewhat inconsistent. At some cache sizes FBR's performance is only marginally better than LRU, and at some other cache sizes its performance is slightly worse (for the fixed set of parameters we chose). However, at about 8M bytes cache size, FBR performs noticeably better than LRU. For example for the HAW trace, FBR's miss ratio at 8M bytes is about 30% whereas LRU's miss ratio is al-

most 32%. The relative improvement measure at this cache size is about 18%. At this same cache size, and for these same fixed parameters, a relative improvement of 27% is achieved for the KNG trace. One reason for these inconsistencies as compared to the other traces may be that the VM traces are much shorter (see Table 3.1). Nevertheless, FBR is seen to offer a significant amount of improvement over LRU near the knee of the miss ratio curve for both VM traces. This is an important result because most systems are configured to operate near the knee of the curve.

3.6. Other Performance Measures

We also used the traces to generate results for various other performance measures related to the FBR algorithm, examples of which are shown in Figure 3.4 and Table 3.4. In Figure 3.4 *Reref(i)* (defined above) is plotted for the CUST11 trace where the new boundary position is fixed at 2M bytes. Also shown is the *occurrence ratio* for each count *i*, which is defined as the fraction of all references that result in a block's reference count being set to *i*. This figure clearly shows the relationship between probablity of rereference and reference count: as the reference count goes up, the rereference probability increases, eventually approaching unity. Note that the rate of increase is initially quite high. This provides an intuitive explanation for the performance gains achieved by the FBR algorithm. On the other hand, note that the occurence ratio drops rapidly for increasing reference counts, which implies that only a few blocks attain high reference counts.

In Table 3.4 results are shown indicating what fraction of blocks are on each count chain (where count > 8 refers to the blocks that are not on a count chain since their count exceeds the maximum set at 8) at various points during the UNIX trace, for a cache size of 4M bytes. Also shown for each count are the fraction of blocks that are replaced with that count. As mentioned earlier in Section 2.4, we see from this table that more than 90% of the time a block with a count of one is replaced, and of the remaining cases, at the end of the trace for example, 4.8 (100-92.1) = 61% of the time a block with a count of two is replaced, etc. Thus, it suffices to maintain count chains up to fairly small values of C_{max} in order to achieve most of the performance gains possible using the FBR algorithm.

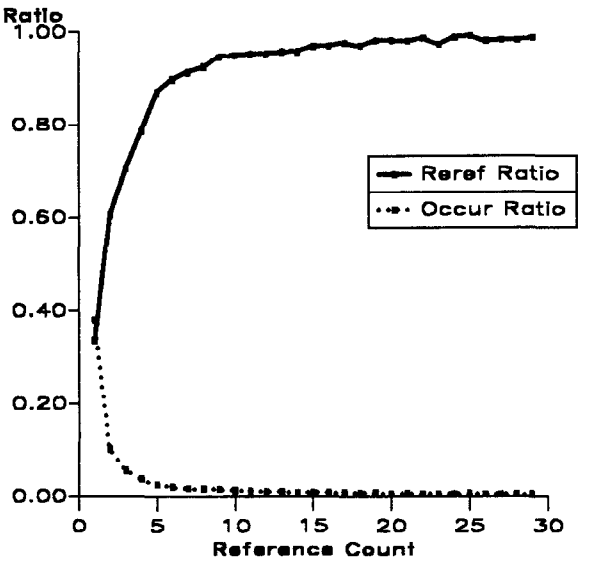

Figure 3.4. *Reref(i)* **and Occurrence Ratios for CUST11**

Chain	After 1/3 trace		After 2/3 trace		At the end trace	
	% on-chain	% replaced	% on-chain	% replaced	% on-chain	% replaced
count 1	38.4	93.0	24.3	92.8	31.8	92.1
count 2	1.1	4.1	0.1	4.4	0.7	4.8
count 3	1.0	1.3	0.1	1.3	0.5	1.4
count 4	0.1	0.4	0.0	0.5	0.0	0.6
count 5	0.1	0.2	3.8	0.3	0.1	0.3
count 6	0.7	0.2	0.9	0.2	0.0	0.2
count 7	0.0	0.1	0.4	0.1	0.2	0.1
count 8	0.0	0.1	5.8	0.1	0.0	0.1
count > 8	58.6	0.5	64.6	0.4	66.7	0.4

Table 3.4. Statistics for Count Chains for UNIX Trace

4. Conclusion

We have presented a new frequency-based replacement algorithm for data caches. This new algorithm factors out locality from reference counts, and thus effectively combines the principles of locality of reference and reference frequency. The implementation complexity and runtime overhead of the new algorithm are only slightly more than the pure LRU algorithm.

Using traces from five different actual systems, block miss ratios for the FBR algorithm were obtained via trace-driven simulations. FBR's miss ratios were compared with those of LRU and OPT, where OPT is an unrealizable optimal algorithm in which the reference string must be known in advance. The results show that FBR can offer a performance improvement of up to 34% over LRU, where the improvement is expressed as a percentage of the difference between the miss ratios of LRU and OPT.

In general, two significant results stand out from our performance study of FBR: (1) the FBR algorithm can successfully factor out locality and can distinguish frequently referenced file blocks from those less frequently referenced in an LRU cache; and (2) the above property can be best utilized at the knee of the miss ratio curve, where the cache size is large enough to permit significant optimization and yet small enough to require a proper utilization of available space.

There are at least two avenues of future research that are suggested by this work. First, the ability of the FBR algorithm to factor out locality and distinguish frequently referenced cache blocks can be applied to virtual memory management when memory mapped file access is provided.

In such systems, file blocks as well as program pages share the same memory space, and there is a need to dynamically restrict the amount of space used by file blocks. Second, an analytic model of the FBR algorithm could be very useful in determining the best set of parameters for the algorithm at a given cache size under given workload characteristics, which currently must be determined experimentally. Since such a model must take into account both locality of reference and reference frequency, which previously have been dealt with separately via stack distance and independent reference assumption models, development of such a unified model represents a significant challenge.

Acknowledgments

We are grateful to G. Bozman for numerous discussions on this work, and for his extensive help in the analysis of the VM traces, and to J. Wolf and K. Smith for their help with the MVS traces.

References

[BOZM89] Bozman, G. P. VM/XA SP2 minidisk cache, *IBM Systems Journal 28*, 1 (1989), 165-174.

[COFF73] Coffman, E. G. Jr., and Denning, P. J. *Operating Systems Theory*, Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[DEVA88] Devarakonda, M. *File Usage Analysis and Resource Usage Prediction: A Measurement-Based Study*, Ph.D. Thesis, Coordinated Science Laboratory, University of Illinois, Urbana, Illinois, 1988.

[DEVA90] Devarakonda, M. Analysis of file cache replacement algorithms using UNIX traces, Res. report RC15410, IBM Research Div., T. J. Watson Res. Ctr., Yorktown Hts., NY, 1990.

[EFFE84] Effelsberg, W., and Haerder, T. Principles of database buffer management, *ACM Trans. Database Systems 9*, 4 (Dec. 1984), 560-595.

[OUST85] Ousterhout, J., Da Costa, H., Harrison, D., Kunze, J., Kupfer, M., and Thompson, J. A trace-driven analysis of the UNIX 4.2 BSD file system, in *Proc. Tenth Symp. Operating Systems Principles*, pp. 35-50 ACM, Dec. 1985.

[SMIT82] Smith, A. J. Cache memories, *Computing Surveys 14*, 3 (Sep. 1982), 473-530.

[SMIT85] Smith, A. J. Disk cache - miss ratio analysis and design considerations, *ACM Trans. Computer Systems 3*, 3 (Aug. 1985), 161-203.