Improved adaptive replacement algorithm for disk caches in HSM systems

Ulrich Hahn, Werner Dilling, Dietmar Kaletta
Zentrum für Datenverarbeitung
University of Tübingen
Germany
ulrich.hahn@zdv.uni-tuebingen.de

Abstract

With an ever increasing amount of data to store, hierarchical storage management (HSM) systems must still use tape for tertiary storage. A disk cache is used to reduce the access time to data stored on tapes in a robot device. Due to the sequential access to tape devices, some HSM systems will transfer whole files between disk cache and tape. In this case the disk cache is forced to handle storage objects of nonuniform data size. In this article the term 'object' is used initially to emphasize that size is a property of the data stored in the disk cache. Thereafter files will be called cache objects and the disk cache will be called object cache.

When dealing with file objects in a HSM system disk cache, size is not the only property that influences object replacement. A new replacement algorithm called ObjectLRU (OLRU) is introduced that considers different object properties for replacement. Using file system traces and cache simulation, the performance of OLRU is evaluated. Compared to the LRU replacement algorithm, the OLRU replacement improved cache hit rates for all simulated cache sizes. The gap between hit rates for the LRU and OPT replacement algorithms, which ranges between 3.2 and 0.7 percent, is reduced to between 1.9 and 0.6 percent. An online optimization of OLRU parameters is used to increase the adaptiveness of the OLRU algorithm by utilizing a genetic algorithm.

1 Introduction

With cheap fast disk storage easily transcending the terabyte range increasingly available, one might think the end of hierarchical storage management (HSM) systems is within reach. Yet the amount of data to be stored grows at a higher rate and requires systems to store even petabytes of data. So we are still confronted with tertiary storage on tape, and disk caches to reduce access times. Due to the sequential access to the tape device, some HSM systems store whole files to tape. Within this setup, files are the smallest logical units of storage, and disk caches have to deal with objects of nonuniform size. The term 'object' is used to emphasize that size is a property of the data stored in the disk cache. Other properties will be introduced later on, at which point files will be called cache objects and the disk cache will be called object cache.

Caching uses locality of reference in data request patterns to increase performance. Reed and Long[1] give an overview of the most commonly used replacement algorithms, namely, LRU, LFU, FBR, FIFO, RAND, and OPT. They show that LRU (Least Recently Used) is the replacement algorithm of choice when caching NFS requests on an NFS server. Within the last few years other replacement algorithms have been introduced, including a compression-based algorithm in Phalke[2] and an application-controlled algorithm in Cao[3][4]. Phalke shows that compression of the IRG (Inter-Reference Gap) can be used to predict replacement and improve perfor-

mance. Cao allows application some control over placement, which can dramatically increase performance, but requires users to have inside knowledge of the cache architecture. While these approaches are based on technical improvements, Pitkow[5] makes use of findings in other scientific disciplines; he uses psychological research results on human memory to improve the performance of WWW caches. The above approaches have one thing in common: they rely solely on access patterns to select objects for replacement. No additional replacement criteria is taken into consideration for influencing system utilization. If a disk cache stores whole files, the size of an object is an additional replacement criterion as different object sizes will require replacing combinations of objects. Smith[6] uses file reference and file size as a combined replacement criterion. He compares several algorithms that use the product of file size and a power of the most recent reference time. He then replaces the file with the maximum product. Although this approach leads to increased hit rates, he attains the best results by exploiting the empirical distribution of inter reference times for different file size ranges (offline optimization).

If we reflect further on HSM systems, additional object properties – e.g., the cost of file allocation – come to mind that influence replacement as well. Therefore, we offer a new replacement algorithm called ObjectLRU (OLRU), which will take into account various object properties. The use of a weighting function to evaluate combinations of objects provides a more flexible approach. To increase adaptiveness, the weighting function is optimized using an online genetic algorithm. The influence of algorithm runtime on cache performance is considered, in order to exclude the possibility of performance loss. The algorithm is based on LRU replacement but incorporates features introduced by Pitkow[5].

Miller[7] and Katz give an in-depth analysis of file migration in HSM systems. They use supercomputer file access traces to drive their simulations. One of their most interesting findings is that migration and replacement algorithms should be optimized for reading. If users access files interactively, they tend to wait for reads, and will evaluate system performance accordingly.

To evaluate replacement performance, trace-driven simulations for a read workload were performed, allowing for different system metrics (see Muntz[8]).

2 Cache algorithms

Caches are used to bridge large gaps between access times to data. Caches are therefore applied on almost every level of the memory hierarchies found in computers. They range from data and instruction caches, found in microprocessors, to disk caches in HSM systems. The common problem for all cache algorithms is to decide which data object will be replaced, and how much time can be spent to make a replacement decision.

Throughout this paper LRU and OPT replacement algorithms will be used to evaluate the performance of the new algorithm. Therefore, we will give a brief summary of their replacement technique. The LRU (Least Recently Used) algorithm will replace the cache object with the oldest reference. For ease of operation, a replacement stack is kept, which is ordered by reference. The OPT (OPTimal) algorithm works much the same way, but will replace the object with the furthest future reference. It is obvious that OPT replacement can only be used if all requests are known in advance. It is therefore called an offline algorithm.

2.1 Performance gain

One important point when designing a cache is to have a closer look at possible performance gain and influencing factors. The performance gain g is defined to be the ratio of access times without and with cache.

$$g = \frac{\text{access time without cache}}{\text{access time with cache}} \tag{1}$$

The influencing factors are bandwidth and latency of I/O devices and the hit rate of the cache. The calculation of the performance gain is mainly used to give an estimate for the influence of the replacement algorithm runtime. It therefore neglects some factors like disk utilization, disk fragmentation, or tape device blocking.

In case of a read, the access time t to data of size s stored on tape can be calculated using the latency time t_{lat} and the device bandwidth bw:

$$t = t_{lat} + \frac{s}{bw} \tag{2}$$

If a disk cache is used, data is buffered on a disk device with lower latency and higher bandwidth. Access

to buffered data takes the cache hit access time t^c_{hit}:

$$t^c_{hit} = t^c_{lat} + \frac{s}{bw^c} \quad (3)$$

using disk latency time t^c_{lat} and disk bandwidth bw^c. The latency time of the disk cache is composed of the hardware latency t^{hw}_{lat} of the disk device and the runtime t^{alg}_{lat} of the cache replacement algorithm.

$$t^c_{lat} = t^{alg}_{lat} + t^{hw}_{lat} \quad (4)$$

Access to data not buffered (miss) requires the data to be read from tape. Depending on the method used to transfer data within the memory hierarchy, it can either be accessed from system memory or through the disk cache. Assuming the first case the miss access time t^c_{miss} will be:

$$t^c_{miss} = t \quad (5)$$

Introducing the hit rate $hitrate$ equations 3 and 5 can be combined to the cache access time t^c

$$t^c = hitrate \times t^c_{hit} + (1 - hitrate) \times t \quad (6)$$

We can decide whether the cache increases system performance by calculating the equal performance hit rate hit_{eq}. Equal performance is given by equal access times t^c and t. Using equations 2 and 6 we get:

$$hit_{eq} = \frac{t^c_{hit}}{t} \quad (7)$$

If the hit rate drops below hit_{eq} the cache will decrease system performance. The performance gain g becomes

$$g = \frac{1}{1 + (hit_{eq} - 1) \times hitrate} \quad (8)$$

The maximum performance gain g_{max} will emerge if the hit rate is 1.

$$g_{max} = \frac{t}{t^c_{hit}} \quad (9)$$

Given the equations above, we can now have a look at the influence of replacement algorithm runtime on cache performance. Table 1 shows approximated latency and bandwidth for the devices used in a HSM system setup.

If we assume hit rates in a real disk cache to be between 50 and 90 percent, the performance gain can be calculated for different algorithm runtimes and data chunk sizes. Table 2 lists maximum performance gain, performance gain at 50 and at 90 percent hit rate for data chunk sizes of one kB, one MB and one GB and algorithm runtime ranging from zero to one hundred milliseconds.

Table 1: Latency and bandwidth for devices in a disk cache setup

type	bandwidth	latency
disk	10 MB/s	10 ms
tape	5 MB/s	30 s
robot	—	10 s

Table 2: Performance gain g_{max}, g(90) and g(50) for different data chunk sizes s and algorithm runtime t^c_{lat}

size s	t^c_{lat}	g_{max}	g(90)	g(50)
1 k	0 ms	3960.4	9.977	1.999
	1 ms	3603.6	9.975	1.999
	5 ms	2649.0	9.966	1.999
	10 ms	1990.1	9.954	1.999
	50 ms	665.6	9.867	1.997
	100 ms	363.3	9.758	1.995
1 M	0 ms	365.45	9.760	1.995
	1 ms	362.16	9.758	1.994
	5 ms	349.57	9.749	1.994
	10 ms	335.00	9.738	1.994
	50 ms	251.25	9.654	1.992
	100 ms	191.43	9.551	1.990
1 G	0 ms	2.400	2.105	1.412
	1 ms	2.400	2.105	1.412
	5 ms	2.400	2.105	1.412
	10 ms	2.400	2.105	1.412
	50 ms	2.399	2.104	1.412
	100 ms	2.397	2.103	1.411

The values of table 2 show the expected behavior for the performance gain. With small data chunk size s the performance gain is determined by access latencies, for large sizes it is ruled by bandwidth. We can also see that

replacement algorithm latency mostly affects the maximum performance gain g_{max}; it decreases as $1/x$ with t^c_{lat}. On the other hand, performance gain with realistic hit rates will drop in an almost linear fashion with increased t^c_{lat}, as is shown for $g(90)$ in figure 1. Even for algorithm runtime of 10 ms the performance loss will be less than 0.3 percent.

In a loaded system, latency and bandwidth of disk devices will increase, so that the performance gain will drop. With increased device latencies the influence of algorithm runtime will further diminish.

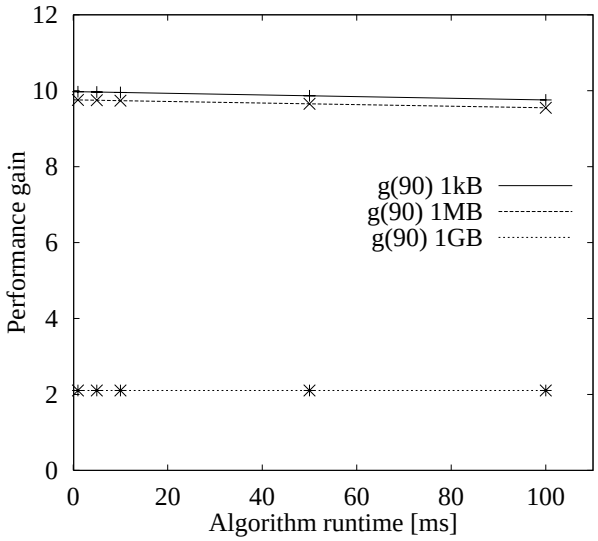

Figure 1: Performance gain $g(90)$ versus algorithm runtime for different data chunk sizes s

With the system setup in table 1 and the simplified calculations on performance gain, the gap in access times for disk and tape devices in a HSM disk cache gives rise to replacement algorithm runtime in the order of 10 ms without noticeable performance loss. This allows the use of more sophisticated replacement algorithms.

2.2 Improved replacement

Well-known replacement algorithms like LRU, LFU, or others use one criterion to decide which cache object will be replaced. These algorithms are easy to realize. Their replacement comes with high hit rates and short replacement times. As replacement for these algorithms is solely based on access patterns, no further object properties are considered.

Advanced replacement strategies are usable as long as replacement algorithm runtime is kept below the limit calculated in section 2.1. The design of the new replacement algorithm is based on the following points:

- adaptiveness
- performance
- reliability
- flexibility, multiple object properties considered
- extensibility

The first step in developing the new algorithm has been to look at cache object properties that will increase the hit rate in an object cache. If we recall that object caches handle data of nonuniform size, the principles of improved operation are easy to understand. The following example will illustrate the basic point of considering object size for replacement, as opposed to standard LRU replacement. Figure 2 shows the bottom of an LRU-ordered object cache. If the cache is completely filled and an uncached object of size six is accessed, cache objects of a minimum combined size of six must be replaced. Using LRU the objects a, b, c and d would be replaced, freeing a total size of eleven. Since we initially needed 6 free units, replacing only object d is sufficient. In this particular case

objectId	size	reference
f	3	10
e	1	11
d	6	16
c	2	17
b	2	24
a	1	27

Figure 2: Tail of the LRU-ordered object cache. Object properties shown are last reference and size

it is even optimal, since the cache fill rate is kept and the three objects not replaced can still be accessed without a miss. For this particular setup we get optimal replacement by replacing exactly one object of the size required and a reference as well included for replacement by LRU. This means that by introducing size we get three criteria for optimal replacement in an object cache, namely, the number of objects, the size of the replaced objects, and their references. The example used simplifies part of the problem, since for actual replacement it is unlikely that one cache object will fit all the criteria. Therefore combinations of objects have to be considered. These ideas can also be applied directly to OPT replacement, and it is obvious that OPT replacement is not optimal for object caches. The OPT replacement algorithm is used to give an upper limit on possible cache performance when comparing different replacement algorithms. So an optimized OPT algorithm for object caches has to be introduced as well. This algorithm will be called ObjectOPT or OOPT for short.

In transfering this description of optimization to mathematical terms, the following optimization criteria K are used. All criteria have global minima, zero if possible. If we assume that all objects at the end of the LRU stack have the same probability of a future access, the probability for a hit remains highest if only a small number n of objects have to be replaced. Replacing a single object is best. The resulting criteria K_n is then given by

$$K_n = n - 1 \tag{10}$$

If a total size s_{rep} has to be freed by replacement, it is best if the sum of replaced objects sizes is equal to s_{rep}, or

$$K_s = \sum_{i=1}^{n} s_i - s_{rep} \tag{11}$$

Under the OPT algorithm, objects with the farthest future reference will be removed. For OOPT the sum of differences to the farthest reference r_{max} has to be minimal.

$$K_r = \frac{1}{n(n+1)} \sum_{i=1}^{n} (r_{max} - r_i + 1) - \frac{1}{2} \tag{12}$$

The reference criteria K_r is scaled by a factor $1/(n^2+n)$ to keep it independent of the number of objects n. This ensures that all criteria are orthogonal and a weighting function W can be defined that is a simple sum of all criteria functions K. The use of orthogonal functions is important for two reasons. The first reason is that criteria can be selected independently to influence replacement behavior (see also the note at the end of this section). The second reason comes into play when optimization is used to determine the constants c, as orthogonal functions will allow the identification of criteria influencing optimal replacement. By applying a constant c to every criteria, the influence of criteria on the weighting function can be adjusted.

$$W = c_n K_n + c_s K_s + c_r K_r \tag{13}$$

To decide which objects are to be replaced, OOPT has to apply the weighting function W to every possible combination of cache objects. The combination with the smallest W will then be replaced. Given p cache objects, meaning 2^p possible combinations, it is obvious that algorithm runtime may well exceed the researcher's lifetime. So a limitation of the number of combinations is necessary. Another point is that OOPT replacement depends on the setting of constants c, and it does not represent a single optimal replacement algorithm, in contrast to OPT. This clearly outlines the problems we are faced using multiple replacement criteria.

Based on the improvements achieved using OOPT replacement, the above techniques were used to build the ObjectLRU (OLRU) replacement algorithm. Since OOPT is an offline algorithm, the OLRU design requires additional runtime issues be taken into account. Furthermore, additional replacement criteria are introduced to increase flexibility and to prove the extensibility of the algorithm. As with OOPT, OLRU replacement is based on using the weighting function W. Two additional replacement criteria are introduced with OLRU. First, the position p in the replacement stack is used for K_p

$$K_p = \frac{1}{n(n+1)} \sum_{i=1}^{n} (p_{max} - p_i + 1) - \frac{1}{2} \tag{14}$$

As can be seen from equation 14 the position criterion closely resembles the reference criterion of equation 12. In fact, the position criterion is a weakened reference criterion, as position is a relative and reference an absolute criteria. So using K_p prevents objects from getting older.

Since K_p and K_r are not orthogonal, only one of them should be used in actual replacement. This is done by setting either c_p or c_r to zero. The second new criterion is cost K_k. Cost should be seen as a more abstract criterion to describe object properties that depend on utilization of system resources. In the case of an HSM system, and therefore in the remainder of this paper, cost shall be considered the time between request and availability of a cache object. The specific feature distinguishing cost from all other criteria is the fact that cost will change every time an object is accessed, depending on actual system usage. Cost is not a property of the cache object itself, but of the state of the whole HSM system. K_k is the sum of cost k of cache objects.

$$K_k = \sum_{i=1}^{n} k_i \tag{15}$$

K_k is also the only criterion with its minimum not equal to zero. The weighting function of OLRU replacement is given by:

$$W = c_n\, K_n + c_s\, K_s + c_r\, K_r + c_k\, K_k + c_p\, K_p \tag{16}$$

(Note: by setting all c expect c_p or c_r to zero, OLRU replacement becomes equal to LRU replacement).

2.3 Algorithm runtime

The first part of this section discussed in depth the influence of algorithm runtime on cache performance. To keep within the limits of unnoticed performance loss when faced with 2^p possible combinations, it is necessary that the actual number of calculated combinations be reduced. At this point different influences lead to a working solution. Psychology and human memory research (see Pitkow[5]) suggest an LRU technique using a window of possible replacement victims. Simple considerations exclude combinations if a size limit is reached, or a better combination is already found. The requirements for building combinations are:

- combinations are build upon a small number of cache objects located in a window at the tail of the LRU stack. The size of this window is determined by the total size to be freed.
- if a combination's total size already exceeds s_{rep} no further cache objects are added to this combination.
- if the weighting function W for any subcombination is worse than W_b of the best combination, no further cache objects are added to this combination.
- if any c is set to zero the corresponding criterion is not calculated.

With large caches these limitations may still be insufficient to reduce the depth d of combinations. In this case, additional techniques can be used to further reduce d, so that a maximum combination depth d_{max} is not exceeded.

- Fallback to LRU replacement:
  Cache objects will be replaced using LRU until only d_{max} objects remain for OLRU replacement.
- Overlapping subwindow replacement:
  OLRU replacement occurs in a subwindow of size d_{max}. Using an overlap u only $d_{max} - u$ objects are replaced; u objects are kept and used as part of the next subwindow. This continues until enough objects are replaced.

Because the subwindow technique applies weighting to all objects and therefore tends to keep valuable objects (this was shown to be true in simulations), it is the technique used within OLRU.

If using all the above techniques results in algorithm runtime below the limit calculated in section 2.1, the design goals of this section are fulfilled by the OLRU replacement algorithm. By introducing the weighting function W the OLRU algorithm considers multiple object properties and can be extended to fit different replacement problems. The ability to adapt to different system requirements is based on the description of criteria for different object properties and tuning of the weighting function W modifying the constants c.

3 Optimization

With the introduction of constants c in the weighting function W of section 2.2, we gain the ability to adapt the replacement policies to the current state of the disk cache. In the early design state this was done by running

simulations (see section 4 for details) and hand-tuning the constants to increase hit rates. This offline technique may only be used to prove that the weighting functions are suitable to improve cache replacement. For real applications the constants of the replacement algorithm have to be tuned at runtime. This is achieved by utilizing genetic algorithm to optimize constants.

3.1 Genetic Algorithm

Genetic algorithms are based on methods of evolution. They have been used in optimization since about 1975 (see Holland[9] and Goldberg[10] for details). Emulating evolution, a population of n individuals is observed whose parameter sets are bit-string coded. After a defined period of time, a generation, the individuals of the population are evaluated using a fitness function. The population is then sorted according to the resulting fitness values. Selection decides which individuals are used for cross-over of bit-strings to generate new individuals, which will replace the individuals with lowest fitness. Mutation of individuals is used to maintain variety in the population.

The genetic algorithm is realized by simultaneously simulating the n individuals. Although this induces an increased use of computational resources, replacement benefits in several ways. First, by increasing the number of individuals n, the parameter space is spanned more effectively. Second, after evolving over several generations, most individuals will represent a reasonable solution, which will in turn increase the probability of finding a good solution, even with rapidly changing replacement patterns. Third, it is possible to reserve one or more individuals for simulating fixed replacement algorithms, e.g. LRU, so that cache performance may never fall behind the performance of reliable algorithms.

The quality of genetic optimization depends mainly on two factors: the fitness function and the duration between generations.

3.2 Fitness function

To get a working fitness function three goals for an optimized replacement are defined:

- obtain a high hit rate, minimize the number of replaced objects
- minimize time spent in the replacement algorithm
- minimize the average cost of replaced objects

The individual goals are derived from considerations in section 2.2. Contrary to the weighting functions, these goals are not orthogonal, they depend heavily on each other. This is exactly the reason why genetic algorithms are applied. If cache performance goals were orthogonal, the influence of constants c on performance could be looked at separately and then solved analytically. As this is not the case, it is up to the genetic algorithm to change parameters in a way that maximizes fitness. The terms above are transformed to calculable terms r of the fitness function f. They are calculated relative to the actual population and scaled in the range $[0 : 100]$. That means if v_i is the value of individual i, v_{max} and v_{min} being the maximum and minimum values of the population, r can be calculated by

$$r = 100 \times \frac{v_i - v_{min}}{v_{max} - v_{min}} \tag{17}$$

The total fitness f is the sum of all terms.

$$f = \sum r \tag{18}$$

The influence of terms r on cache performance can be determined by deselecting terms for simulation.

3.3 Algorithm details

The implemented genetic algorithm simulates a population of 32 individuals. The constants c of section 2.2 are converted to 16-bit bit-strings, using defined offsets and multipliers. With every generation the 8 individuals with lowest fitness will be replaced. To generate new individuals, parents for cross-over are selected by a roulette method. Each individual is assigned a number of slots depending on its fitness. The fittest individual will get n slots, while the least fit individual will get 1 slot. Then slots for the parents are selected randomly, as is position and length of cross-over. After cross-over is performed, individuals are selected for mutation with a defined probability. Mutation will flip up to 16 bits at a random position of the bit-string. Duplicate individuals produced by cross-over or mutation will be discarded.

As replacement time is critical to cache performance, it is obvious that running 32 additional simulations would override the benefits of improved replacement. Therefore the genetic algorithm is built to be used as a module independent of the caching algorithm. The caching algorithm submits all access requests, with accompanying metadata, to the genetic algorithm and in turn requests constants c. With this technique it is possible to split optimization and replacement tasks. Optimization can even be dedicated to another machine, thus reducing the use of system resources for the disk cache. With this approach care has to be taken to keep metadata synchronized.

4 Simulation

All results in this paper are based on cache simulation. The simulation program is designed to deliver reliable results for cache performance without implementing a complete HSM system. Cache performance gain g is determined by hit rate $hitrate$ and algorithm runtime t^c_{lat} given by equation 8. The obtained hit rate depends on the replacement algorithm, cache size, and the locality of reference for cache requests. Algorithm runtime depends on algorithm coding, system computational resources, and the structure of cache data and metadata. The simulated algorithm will deliver hit rates, while the simulation program will deliver algorithm runtime. To obtain usable simulation results, several requirements must be fulfilled:

- Replacement algorithms are completely coded and runtime optimized.
- Modular design of replacement algorithms allows easy simulation and optimization of the new algorithm.
- Input data to simulation (requests) must resemble a real workload.
- As system performance will be rated by read performance, a read workload is used.

4.1 The simulation program

The internal structure of the cache will determine algorithm runtime. The way to efficiently organize a disk cache is to use a fully associate cache. Without a proper object-address or object-ID scheme, searching among n cache objects will require O(n) comparisons. The simulation program therefore uses an AVL-tree, as described by Wirth[11], to store object metadata; this will guarantee O($\log n$) search comparisons. Metadata for OLRU cache objects consist of the object-ID, last reference, size, pointer to data, cost and two additional pointers to build a double-linked list realizing a LRU-stack.

As simulation need not handle file data, neither data storage nor data transfers are coded. The cache is divided into blocks to resemble block based disk storage. Although it is likely that, for performance reasons, files in a HSM disk cache will be stored unfragmented, the simulated cache will distribute file data and use all available blocks. This will ensure that the replacement algorithm will not interfere with file allocation issues on a real disk. For practical reasons the number of combinations in OOPT replacement was reduced by the means introduced with OLRU.

The reliability of results obtained by simulating a cache depends on the quality of request patterns used. Artifical request patterns tend to be unable to reproduce locality of reference found in real systems. So the recorded request traces of a real file system were used, as is common in cache simulation (see Muntz[8], Miller[7] or Smith[6]).

4.2 Input traces

The traces used are the public available traces of the Sprite file system, as described in Hartman[12]. The Sprite traces consist of all file system requests obtained within a period of eight days. File system request were recorded for a network of approximately 50 workstations.

Since the Sprite traces' requests do not contain any information on request cost, this information had to be generated artificially. While no cost information is included, each request is identified by the machine-ID of the host that dispatched the request. For each request, cost is calculated using data size and an assigned latency and bandwidth depending on the machine-ID, as is shown in table 3. Using the Sprite traces library functions, file size for valid requests can be limited. This is used to avoid requests for files larger than the total cache size.

Table 3: Latency and bandwidth set for machines of the Sprite traces

machine-ID	bw [MB/s]	t_{lat} [s]	device
1 - 20	5	0.1	tape
21 - 40	1	0.2	net
41 - 60	10	0.01	disk
61 - 80	2	4	MO-robot
81 - 100	5	10	tape-robot

Due to the amount of requests in the traces, the total number of requests was limited to 100 000. Recording of hits starts when the cache fill rate reaches 90 percent to exclude warm up effects.

Simulations were executed on a Pentium 166 computer with 48 MB memory running Linux 2.0.33. Average simulation runtime is 20 to 30 minutes.

4.3 Evaluating cache performance

To evaluate cache performance several metrics are considered:

- hit rate, given by

$$\frac{\text{number of hits}}{\text{number of requests}} \tag{19}$$

- average replacement cost

$$\frac{\text{cost of replaced objects}}{\text{number of replaces}} \tag{20}$$

- replacement time

$$\text{runtime spent in replacement algorithm} \tag{21}$$

Hit rate is the most important metric for describing cache performance. It is clearly the most influential factor for cache performance gain (see equation 8). The hit rate describes the ability of the replacement algorithm to adapt to request patterns. Another metric of interest is replacement cost. As total replacement cost increases with the number of replaced objects, it depends strongly on the replacement hit rate. Therefore, the average replacement cost will be considered, as it describes the ability to keep costly objects, thus conserving system resources. The third metric of inerest is cache replacement time. By using equations 4 and 8, replacement time can be used to calculate the induced performance loss.

5 Results

First of all simulations were performed using LRU and OPT replacement algorithms, in order to become familiar with the Sprite traces. Figure 3 shows the hit rates obtained for a cache block size of 1 kB and total cache size ranging from 8 kB to 1 MB. The attained hit rates indicate a high locality of reference found in the Sprite traces. The maximum cache size was therefore limited to 1 MB. Even though this is a small size compared to disk cache sizes, it allows examination of principles of replacement strategies. Improved replacement does not depend on total cache size, as only a small number of cache objects is affected. Or, to use a metaphor from electrical engineering, the information contained in an AC signal is not

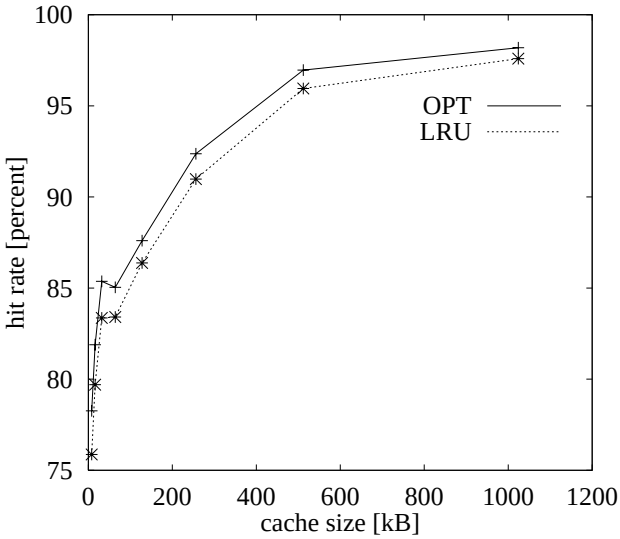

Figure 3: Hit rates versus cache size for LRU and OPT replacement algorithms

influenced by a large DC component. So the results can be applied to large disk caches, even if the size of the simulation cache is small.

Despite higher hit rates, LRU and OPT curves can be compared to the results of, e.g., Reed[1]. The slight impact at a cache size of 64 kB can be seen as a result of whole-file caching. Figure 4 shows the file size distribution within the Sprite traces. While the number of files

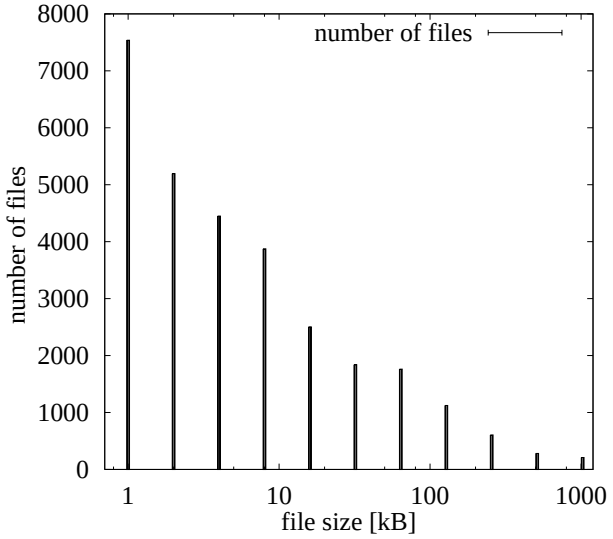

Figure 4: File size distribution of the Sprite traces

decreases with file size, there is a noticeable step at 64 kB. This step will lead to increased replacement of large-sized chunks and therefore to a lower hit rate.

To achieve an upper limit for the object-cache hit rate, OOPT simulations were performed. Figure 5 shows the improvement of OOPT hit rates over OPT replacement. The weighting function of OOPT was tuned for the Sprite traces by adjusting c. Compared to OPT replacement improved hit rates are attained for all cache sizes. The improvement is most evident at 64 kB cache size, when large-sized chunks have to be replaced. In spite of limited combination depth (see section 4.1) and a hand-tuned weighting function, the OOPT algorithm clearly increases cache performance by means of hit rate. OOPT will be used as an upper limit for cache hit rates, if object-cache replacement algorithms are compared.

Now that we have seen that the OOPT algorithm improved offline cache replacement for real workloads, we

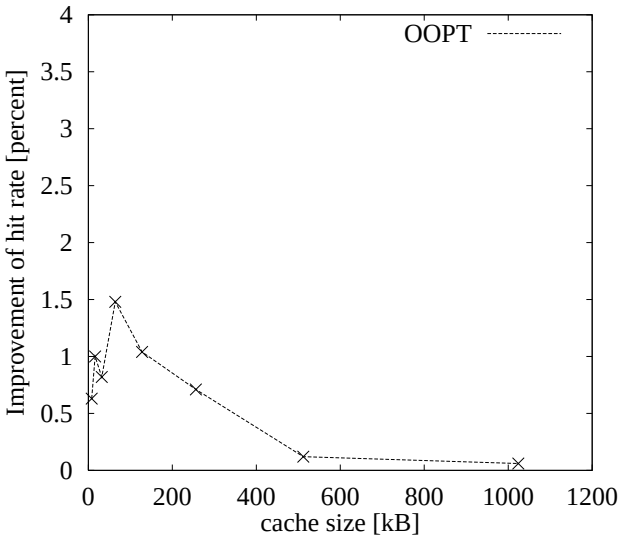

Figure 5: Improvement of the OOPT replacement algorithm's hit rates over OPT replacement

can have a look at OLRU replacement to see if these improvements can be replicated. Figure 6 shows the hit rates for OLRU compared to OOPT and LRU replacement. The performance gain is obvious, although the absolute increase seems to be small. If LRU replacement hit rate is used as a lower limit, we can also calculate the improvement of hit rates. The resulting curves are shown in figure 7. This figure includes the hit rates of OPT replacement to show that OLRU replacement algorithms can outperform the 'former optimal' OPT algorithm at cache sizes of 64 kB and 128 kB. Hit rate improvements of OLRU compared to LRU replacement range from 0.1 to 1.8 percent. OLRU replacement is most effective compared to LRU if large data chunks are replaced. With these increased hit rates we can have to look at the algorithm runtimes of figure 8. Algorithm runtime was measured using the gettimeofday() system call, since no other high precision timers were available. The time resolution of the gettimeofday() system call is below one μs, as Linux uses the Pentium cycle counters for timing information. The disadvantage of using the gettimeofday() system call is that total time is measured instead of process time. Figure 8 shows the runtime distribution, giving the percentage of runtimes below 0.1 ms, 1 ms, 3 ms, and 5 ms. Runtimes above 5 ms were less then 0.2 percent for all cache

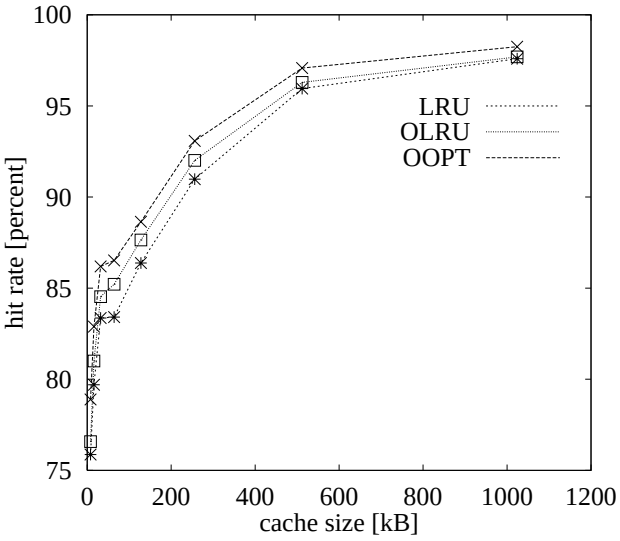

Figure 6: Hit rates versus cache size for OLRU, OOPT and LRU replacement algorithms

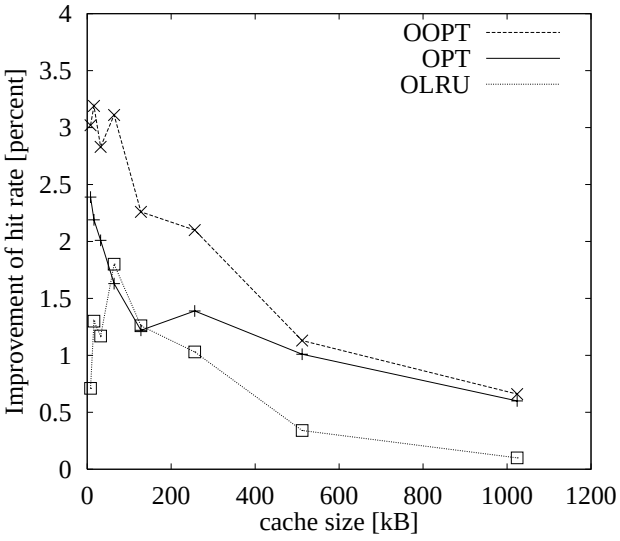

Figure 7: Improvement over LRU hit rates for the OLRU and OPT replacement algorithms

sizes. Ninety percent of all runtimes are less than 1 ms. With cache sizes of 256 kB and higher runtimes greater than 1 ms gain a share of up to 9 percent.

But not only hit rate can be used to estimate cache performance. In section 4.3 further metrics were intro-

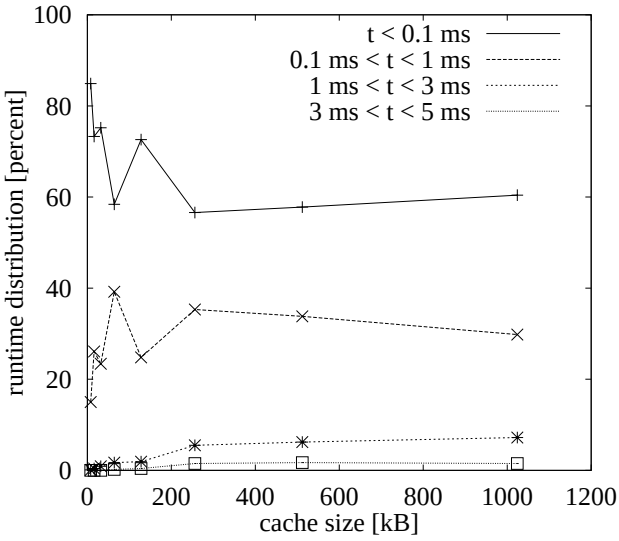

Figure 8: Runtime distribution versus cache size for OLRU replacement

duced, which will be considered now. Replacement time will not affect performance, if it is kept below the limit assumed in section 2.3. The replacement times observed in simulations show no significant relation to changed weighting function criteria c. This may be related to the fact that absolute time is measured instead of process time. The measured time will therefore include additional system process time if the simulation program is scheduled by the kernel. With the current setup, replacement time cannot be used as a metric to characterize cache performance. It can, however, be used with the genetic algorithm's fitness function to trigger a penalty if a timing limit is transgressed.

With regard to replacement cost, a strong dependency between it and hit rate can be observed. The source of this dependency is obvious: more hits mean lower replacement cost since fewer objects are to be replaced. Therefore, average replacement cost was chosen as a metric, to see if a more cost-efficient replacement can be achieved, while keeping hit rates high. Figure 9 compares the average cost of replaced cache objects for OLRU and LRU algorithms. The average replacement costs for OLRU replacement are slightly lower then for LRU replacement. So the OLRU algorithm not only increases hit rates, but at the same time decreases average cost. This

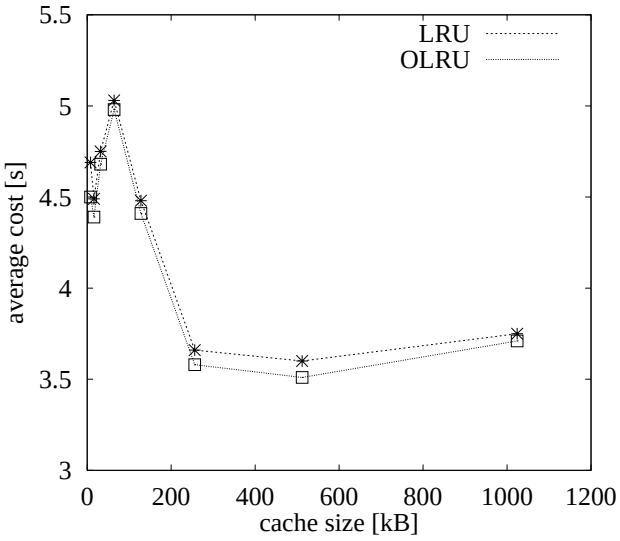

Figure 9: Average cost of replaced objects versus cache size comparing OLRU and LRU replacement

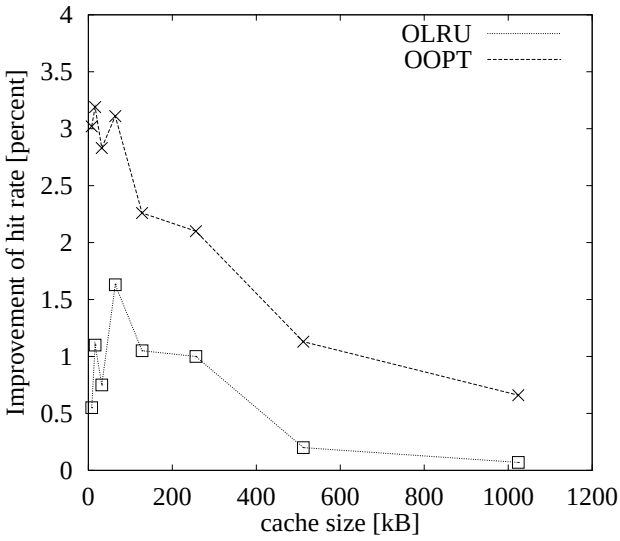

Figure 10: Improvement over LRU hit rates for online optimized OLRU replacement

result proves that OLRU can indeed be used to improve more than one replacement criteria.

Last but not least, we will now discuss how OLRU replacement can be optimized using the genetic algorithm. Figure 10 shows the improvement over LRU hit rates for online optimized OLRU replacement. Though hit rates for optimized OLRU are improved, online optimized OLRU will not meet the hit rates attained by hand-tuned OLRU. This is indicated by the following points.

- Performance will usually drop with an increased generation period.
- Performance will decrease with cache size, meaning a smaller number of total generations.
- For a small cache size with a large number of generations, online optimized OLRU performs as well as hand-tuned OLRU.

All these points indicate that the number of requests generated by the Sprite traces is too small to be efficiently used for online optimization. The first two points are a result of a decreased number of generations and thus less evolved solutions. Warm up effects may occur as well. At the beginning of simulation, all solutions except one are randomly initialized, so the number of 'advanced' solutions will be limited for some generations. Despite these restrictions, online optimization showed the ability to adapt to Sprite traces without manual interference.

6 Conclusions and future work

The OLRU replacement algorithm introduced in this paper proves to be an expandable, well performing replacement algorithm for object caches. The simulations showed improvement of hit rates between 0.1 and 1.8 percent compared to LRU replacement. Most noticeable improvements occur when large data chunks are replaced. The measured replacement algorithm runtime suggests that the influence on cache performance can be neglected, as runtime is kept below the limit calculated for a disk cache.

To increase the adaptiveness of the OLRU algorithm an online optimization was introduced using a genetic algorithm. The online optimized OLRU replacement shows improvements over LRU, but performs slightly worse than hand-tuned OLRU. The performance drop is believed to be based on the limited number of requests available in the Sprite traces used as simulation input. To improve online optimization it is necessary to use more suitable

input traces. The traces would be perfect if they included data on object cost and system workload, as this allows us to decide if replacement cost can be used to evaluate cache performance by means of overall usage of system resources.

The flexibility introduced by using a weighting function can be used to include even application induced replacement criteria. This will allow the results of Cao[3][4] to be integrated. To further improve cache performance other methods, such as prefetching (see Bennett[13]), should be considered as well.

References

[1] B. Reed, D.D. Long, Analysis of caching Algorithms for distributed file systems, Operating Systems Review, Vol. 30, No. 3, 1996

[2] V. Phalke, Modeling and Managing Program References in a memory hierarchy, Ph.D. Thesis, State University of New Jersey, New Brunswick, 1995

[3] P. Cao, E. Felden, K. Li, Implementation and performance of application-controlled file caching, Proceedings of the first USENIX symposium on operating system design and implementation, 1994

[4] P. Cao, E. Felden, A. Karlin, K. Li, Implementation and performance of integrated application-controlled caching, prefetching and disk scheduling, Proceedings of the ACM SIGMETRICS conference on measurement and modeling of computer systems, 1995

[5] J.E. Pitkow, M.M. Recker, A simple yet robust caching algorithm based on dynamic access patterns, In Proceedings of the Second World-Wide Web Conference, Amsterdam, Elsevier, 1994

[6] A.J. Smith, Long Term File Migration: Development and Evaluation of of Algorithms, CACM, Vol. 24, No. 8, 1981

[7] E.L. Miller, R.H. Katz, An Analysis of File Migration in a UNIX Supercomputing Environment, Proceedings of 1993 Winter USENIX, 1993

[8] D. Muntz, Multilevel caching in distributed file systems, Ph.D. Thesis, University of Michigan, 1994

[9] J.H. Holland, Adaption in natural and artifical systems, University of Michigan Press, Ann Arbor, Michigan, 1975

[10] D.E. Goldberg, Genetic Algorithms in Search, Optimization, and Machine Learning, Addison-Wesley, Reading, Massachusetts, 1989.

[11] N. Wirth, Algorithms and Data Structures, Prentice-Hall, Englewood Cliffs, 1986

[12] J.H. Hartman, Using the sprite file system traces, University of California, Berkeley, 1992

[13] J. Bennett, M. Flynn, Reducing cache miss rates using prediction caches, Technical report CSL-TR-96-707, Stanford University, 1996