New Results on Generalized Caching *

Saied Hosseini-Khayat †

Washington University in St. Louis

Abstract. *We report a number of new results in generalized caching. This problem arises in modern computer networks in which data objects of various sizes are transmitted frequently. First it is shown that its optiaml solution is NP-complete. Then we explore two methods of obtaining nearly optimal answers based on the dynamic programming algorithm that we provided in [5]. These methods enable a trade-off between optimality and speed. It is also shown that LFD (the longest forward distance algorithm which is the optimal policy in the classical case), is no longer optimal but is competitive. We also prove that LRU remains competitive in the generalized case. This is an extension of a famous result by Sleator and Tarjan [12] on LRU. Finally, it is confirmed in the general case that prefetch does not reduce the total cost if "cost" reflects only the number of bytes transmitted.*

Key words: Generalized caching, optimal replacement, replacement algorithm.

1. Introduction

Caching is an effective performance enhancement technique that has been applied in computer systems for decades. A cache, in general, is a fast and small block of storage located between a consumer and a main store of data, and its purpose is twofold: to provide fast access to frequently requested data, and to reduce the volume of data transmitted between the main store and the consumer. To minimize the price/performance ratio, computer system designers take advantage of a hierarchy of caches. This hierarchy starts at the top (i.e. the smallest and fastest) level with CPU registers and includes the on-chip and on-board caches, the main memory and the hard disk. Cached objects in these levels are uniform-size chunks of data, e.g. pages and cache words. The clever management of these caches has been the subject of extensive research in the past decades [13, 14].

The phenomenal growth of the Internet and emergence of distributed information systems, such as the World Wide Web, has necessitated the extension of the cache hierarchy

*Technical Report WUCS-96-25

†Applied Research Laboratory, Department of Computer Science, Washington University, Campus Box 1045, One Brookings Drive, St. Louis MO 63130-4899. Email: saied@arl.wustl.edu. Tel:(314) 935-4460.

into the computer network. As an example, the hard disk now performs as a client-side cache holding requested WWW documents for possible future use. Caching proxy servers form the next level of hierarchy as caches of WWW documents requested by groups of users. Study of regional and national network caches is also underway [8]. These two levels will cache frequently transmitted documents on regional and national scales to reduce traffic load of the network .

The ongoing information revolution is creating new services such as video-on-demand, distributed image archives and on-line libraries, that will also benefit from caching. For example, in the area of video-on-demand, caching of video programs in neighborhood servers is discussed in [9, 11]. Also the use of cache in a distributed image database is discussed in [15].

In many new applications the same scenario occurs again and again: variable-sized data objects *as a whole* are requested, transmitted and cached. Also the performance cost of cache misses are not all identical. For example, if we try to minimize the number of bytes transmitted, then the cost of a miss is (nearly) proportional to size of the missing object. The same is true if user-perceived latency is to be minimized. In addition, when other factors such as the distance an object travels in the network, and the time variation of cost per byte are taken into account, "cost" can become a complicated function of size, time and so on. This motivates a fresh study of the page replacement problem when the assumption of uniformity of size and cost is removed. In this paper, we are interested in the off-line version of this problem, in which the sequence of requests is known in advance. As in the classical case, the study of this version is important from a theoretical standpoint because its optimal solution sets an upper bound on the performance of all other solutions. The classical page replacement problem was optimally and efficiently solved by Belady [2]. In this paper after introducing our notations in Section 2 and defining the generalized problem in Section 3, we show in Section 4 that Belady's theorem does not apply to the generalized problem. Then in Section 5 we prove that this is NP-complete. An optimal method that is based on dynamic programming is presented in Section 6. In Section 7 we propose and explore two methods of finding nearly optimal solutions that allow considerable savings in computation time. Section 8 contains a competitive analysis of Belady's method (LFD). Section 9 presents an extension of a well-known result on prefetching of items. Finally in the last section we summarize our contributions.

2. Notations and Assumptions

Given is a finite universe of n cachable objects $\mathbb{U}$ identified by $\{1, 2, \ldots, n\}$. For each object i there is a positive size a_i and a positive cost c_i. A *cache* at any time contains a set $\mathcal{B} \subset \mathbb{U}$ such that

$$\sum_{i \in \mathcal{B}} a_i \leq B \ ,$$

where positive B is the capacity of the cache. Also it is assumed that $B \geq \max\{a_i\}$. In practice, size and cost are discrete quantities and hence expressible by integers. However we do not need that assumption.

A *sequence of requests* ρ is denoted by $\sigma_1, \sigma_2, \ldots, \sigma_m$, where $\sigma_k \in \mathbb{U}$ for all discrete times k. A sequence ρ has a *maximum cost*

$$W_{max}(\rho) \triangleq \sum_{k=1}^{m} c_{\sigma_k} \ ,$$

which is incurred when caching is not done.

The *state* of a cache is determined by the set of objects it contains and changes in response to requests as a result of the caching algorithm . We denote the state of a cache at time k by $\mathcal{B}_k$, and the *state space* of a cache by

$$\mathbb{B} = \{\mathcal{B} \subset \mathbb{U} \mid \sum_{i \in \mathcal{B}} a_i \leq B\}.$$

Note that $\emptyset \in \mathbb{B}$. The set $\mathbb{B}_j$ denotes the collection of all states containing element j. The *state sequence* $\mathcal{B}_0, \mathcal{B}_1, \ldots, \mathcal{B}_m$ denotes consecutive states of the cache in response to a sequence of m requests, where $\mathcal{B}_0$ is the initial state and $\mathcal{B}_m$ is the final state.

A *caching algorithm (policy)* A takes a request sequence ρ and a cache of size B in initial state $\mathcal{B}_0$, produces a state sequence $\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_m$ and incurs a cost $W(A, \rho, B)$ or equivalently $W(\{\mathcal{B}_k\}, \rho, B)$ which is defined later. The *miss index* (or normalized cost) defined as

$$M(A, \rho, B) \triangleq \frac{W(A, \rho, B)}{W_{max}(\rho)}$$

is the analog of the classical *miss rate* (the number of misses divided by the number of requests). Note that $0 \leq M \leq 1$.

We consider a single cache with non-modifiable non-dividable cache objects, and assume that requests to the cache must be served in the order in which they arrive and every missing item is loaded into cache at the time of request. The penalty of a miss is equal to the cost of the missing item. A hit has no cost. When a new item is loaded, one or more items may have to be purged. Purging involves no cost.

Having introduced our notations, now we move on to the statement of problem in the next section.

3. Problem Statement

Caching is in fact a discrete optimization problem in which each problem instance is mapped to a discrete solution space, and a solution that minimizes an objective function is sought.
Problem Instance: A universe $\mathbb{U}$ of items. For each item $i \in \mathbb{U}$ a positive size a_i and a positive cost c_i. A cache with size B in initial state $\mathcal{B}_0$. A sequence $\rho = \sigma_1 \sigma_2 \ldots \sigma_m$ of items in $\mathbb{U}$.
Solution Space: All state sequences $\{\mathcal{B}_k\}_{k=1}^{m}$ such that

I. for all $i = 1, 2, \ldots, m$, we have

$$\sum_{j \in \mathcal{B}_i} a_j \leq B,$$

II. and for all $i = 1, 2, \ldots, m$, we have

$$\mathcal{B}_i = \begin{cases} (\mathcal{B}_{i-1} - \mathcal{E}_i) \cup \{\sigma_i\} & \text{if } \sigma_i \notin \mathcal{B}_{i-1} \\ \mathcal{B}_{i-1} & \text{if } \sigma_i \in \mathcal{B}_{i-1} \end{cases},$$

where $\mathcal{E}_i \subset \mathcal{B}_{i-1}$.

Objective: Find a state sequence such that its cost $W(\{\mathcal{B}_k\}, \rho, B) = \sum_{k=1}^{m} \delta_k \ c_{\sigma_k}$ is minimized, where

$$\delta_i \triangleq \begin{cases} 0 & \text{if } \sigma_i \in \mathcal{B}_{i-1} \\ 1 & \text{if } \sigma_i \notin \mathcal{B}_{i-1} \end{cases}.$$

Condition *I* reflects the limited capacity of the cache whereas Condition *II* places restriction on the way cache state changes. We have assumed that its state may change only when a miss occurs. In that case the missing item is loaded and some items may have to be purged to satisfy the first condition. A state sequence satisfying the objective is an *optimal state sequence*. An *optimal algorithm* is one which produces an optimal state sequence for all problem instances. In general, if an algorithm requires exact information about all future requests, it called *off-line*. Alternatively, if it requires no advance knowledge of future, it is called *on-line*. The latter category is more useful, because in practice requests are presented to the algorithm one at a time and an immediate response is expected. However, the former can also be useful in applications in which requests are scheduled in advance. It is easy to realize that no on-line algorithm can be optimal in the above-defined sense. We focus on off-line algorithms in this paper.

4. LFD is not Optimal

A special case of the generalized caching problem (GENCACHE), known as the paging or page replacement problem, has a well-known optimal solution. If all items have unit size and cost, then it has been shown [2, 7] that replacing, among all items currently in cache, the item whose next request comes furthest in future is the optimal policy. We call this the longest forward distance (LFD) algorithm or policy. Unfortunately, when either the sizes or costs are nonuniform this policy is not optimal. We demonstrate this by examples.

Case 1. Consider items $\{1, 2, 3, 4\}$ with uniform sizes and non-uniform costs $1, 1, 5, 10$ respectively. Let the cache capacity be $B = 2$. Take the request sequence $\rho = 1, 4, 3, 2, 1, 4$ and start with an empty cache. LFD generates the following state sequence:

$$\text{LFD: } \emptyset, \{1\}, \{1, 4\}, \{1, 3\}, \{1, 2\}, \{1, 2\}, \{1, 4\}$$

and incurs $W(LFD, \rho, B) = 27$. Suppose a different policy A generates the following state sequence:

$$\text{A: } \emptyset, \{1\}, \{1, 4\}, \{4, 3\}, \{4, 2\}, \{4, 1\}, \{4, 1\}.$$

This is a valid state sequence, satisfies ρ and incurs a cost $W(A, \rho, B) = 18$ which is less than that of LFD. This happened because LFD dutifully purged a costly item instead of a less costly item.

Case 2. Consider items $\{1, 2, 3, 4, 5\}$ with uniform costs and non-uniform sizes $1, 1, 1, 3, 3$ respectively. Take the request sequence $\rho = 2, 1, 3, 2, 4, 2, 5, 5, 4, 2, 1, 1, 3$. Let the cache capacity be $B = 6$ and start with empty cache. LFD generates the following state sequence:

$$\text{LFD: } \emptyset, \{2\}, \{1, 2\}, \{1, 2, 3\}, \{1, 2, 3\}, \{1, 2, 3, 4\}, \{1, 2, 3, 4\},$$

$$\{5, 4\}, \{5, 4\}, \{5, 4\}, \{5, 2\}, \{5, 2, 1\}, \{5, 2, 1\}, \{5, 2, 1, 3\}$$

and incurs $W(LFD, \rho, B) = 8$. Suppose a different policy A generates the following state sequence:

$$\text{A: } \emptyset, \{2\}, \{1, 2\}, \{1, 2, 3\}, \{1, 2, 3\}, \{1, 2, 3, 4\}, \{1, 2, 3, 4\},$$

$$\{1, 2, 3, 5\}, \{1, 2, 3, 5\}, \{1, 2, 3, 4\}, \{1, 2, 3, 4\}, \{1, 2, 3, 4\}, \{1, 2, 3, 4\}$$

which is a valid state sequence and satisfies ρ. This policy incurs a cost $W(A, \rho, B) = 6$ which is less than that of LFD. This happened because LFD dutifully removed two items which were farthest in future in favor of one big item.

In both cases it was shown that LFD does not produce the optimal state sequence. The same statement is obviously true in the more general case of items with non-uniform sizes and non-uniform costs. It is natural to look for a simple modification of LFD that fixes its flaws. This, however, turns out to be tremendously difficult. In this paper, we will present an optimal algorithm that has no similarity to LFD and solves the general case. Although this sounds good, the simplicity and efficiency of LFD is lost. The following result suggests that we should not expect to find an efficient algorithm for GENCACHE.

5. Intractability

We showed that LFD does not optimally solves (GENCACHE). It turns out that GENCACHE is NP-complete. We prove this by reducing from the 0-1 Knapsack Problem which is NP-complete [4, Problem MP9]. Specifically, we show that if an efficient (i.e. polynomial time) solution for GENCACHE exists, it can be used to solve the knapsack problem efficiently.

GENCACHE was defined in Section 3. The statement of the knapsack problem is as follows.

Definition 5.1 (Knapsack) INSTANCE: *A set of items $\mathbb{A} = \{1, 2, \ldots, n\}$, for each $i \in \mathbb{A}$ a size $a_i > 0$ and a cost $c_i > 0$. A positive number K (knapsack size) such that $K < \sum_{i=1}^{n} a_i$.*
PROBLEM: *Find a set $\mathcal{K} \subset \mathbb{A}$ such that $\sum_{i \in \mathcal{K}} a_i \leq K$ and $\sum_{i \in \mathcal{K}} c_i$ is maximum.*

The problem is that of finding items, from a given set, that together can fit in a knapsack and have the most total value. Generalized caching, on the other hand, is the problem of finding the best sequence of removals from a cache. The latter can be used to solve the former problem as shown next.

Theorem 5.1 GENCACHE *is NP-complete.*

Proof. First it must be shown that GENCACHE is in NP. Recall that a problem should be stated as a decision problem before it can be said it is or it is not in NP. Therefore we ask that given an instance of GENCACHE whether there exists a state sequence $\mathcal{B}_0, \mathcal{B}_1, \ldots, \mathcal{B}_m$ that satisfies Condition *I* and *II* and its cost $\sum_{i=1}^{m} \delta_i \; c_i$ is less than a given number β. It is easy to see that this can be verified in polynomial time for any given state sequence. Therefore GENCACHE is in NP.

Next we need to construct a polynomial time reduction from KNAPSACK to GENCACHE. Any given instance of KNAPSACK can be mapped into an instance of GENCACHE in the following way: Let $\mathbb{U} = \mathbb{A} \cup \{\xi\} = \{1, 2, \ldots, n, \xi\}$, where ξ is an auxiliary item. Preserve the size and cost of items. Set the size of ξ to $a_\xi = \sum_{i=1}^{n} a_i - K$ and its cost to 1 (arbitrary). Set the cache size to $B = \sum_{i=1}^{n} a_i$ and let it initially contain $1, 2, \ldots, n$. Set the request sequence to $\rho = \xi, 1, 2, \ldots, n$. Now suppose algorithm G solves GENCACHE in polynomial time. The G algorithm on this instance of GENCACHE produces $\mathcal{B}_1^*, \mathcal{B}_2^*, \ldots, \mathcal{B}_{n+1}^*$.
Claim: *The set* $\mathcal{K} = \mathcal{B}_1^* - \{\xi\}$ *is a solution of the knapsack problem.* Proof: In the first step G must (by definition of GENCACHE) place ξ in the cache, i.e. $\xi \in \mathcal{B}_1^*$. Since the cache is initially full, it must remove one or more of items $1, 2, \ldots, n$. The request sequence ρ is such that any item(s) removed from the cache in the first step must be loaded once again. Also when the first miss after ξ occurs, G must remove ξ because there is no room for the missing item. (Otherwise it shouldn't have been removed, because it increases the total cost, so does removing any item other than ξ.) Once ξ is removed, no other items need to be removed until the end. Therefore G misses exactly once on any item that it removes from $\mathcal{B}_0$ and incurs a total cost exactly equal to $W = 1 + \sum_{i \in \mathcal{E}_1} c_i$, where $\mathcal{E}_1$ is the set of elements removed from $\mathcal{B}_0$. The first term in W is unavoidable. Therefore G must minimize $\sum_{i \in \mathcal{E}_1} c_i$. Note that every element in $\mathcal{B}_0$ will be requested once. Therefore this sum is minimized if and only if the total cost of element carried from $\mathcal{B}_0$ to $\mathcal{B}_1^*$ is maximized. Hence G must choose a set $\mathcal{K} \subset \mathcal{B}_0$ to carry to $\mathcal{B}_1^*$ in such a way that $a_\xi + \sum_{i \in \mathcal{K}} a_i \le B$ and $\sum_{i \in \mathcal{K}} c_i$ is maximized. Stated equivalently, G must choose $\mathcal{K} \subset \mathbb{A}$ in such a way that $\sum_{i \in \mathcal{K}} a_i \le B - a_\xi = K$ and $\sum_{i \in \mathcal{K}} c_i$ is maximized. Therefore G must solve the knapsack problem in the first step and $\mathcal{B}_1^* - \{\xi\}$ is the solution. This proves the claim. Now note that by assumption, G solves GENCACHE in polynomial time, therefore it solves the knapsack problem in polynomial time. Since KNAPSACK is NP-complete, then so must be GENCACHE. □

We conclude that GENCACHE cannot be solved in polynomial time unless it is proved that $P = NP$—a very unlikely event. Proving that a problem is NP-complete, however, is not the end of the story. It only means that its exact solution is not scalable. This may be acceptable in applications that only involve small problem instances. In addition, an exact solution may reveal structure that can be exploited in developing its approximate solution. Finally, some special cases of the problem may have efficient solutions, in which case it is better to develop solution for the special case.

We mention that KNAPSACK lends itself to a dynamic programming solution [4, page96]. In technical report [5], we presented an optimal solution dynamic programming for GENCACHE.

6. Approximation

The optimal algorithm in [5] gives an insight on developing approximate methods that are useful when finding the optimal is too costly. Here we investigate two methods.

6.1. Breadth Limiting Method

Recall from the previous section that the catch in the optimal algorithm is the potentially huge breadth of its DAG. Intuition tells us that, instead of expanding every node in each layer, if we expand only up to N best (least cost) nodes and omit the rest, the final result will be close to optimal. This will shorten running time at the cost of optimality. Thus we add the following step between step 3 and 4 of the optimal algorithm:

$3'$. *Omit all but N nodes that have the least cost.*

We implemented the optimal as well as this modified algorithm and performed a sets of experiments. A universe of 20 items with sizes 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 and costs 2, 2, 5, 5, 5, 10, 10, 10, 10, 10, 3, 3, 3, 4, 4, 4, 5, 5, 5, 5, respectively, was picked. The cache size was fixed at $B = 30$. Two random traces of 1000 requests was generated in the following ways:

1. Non-localized Trace. For each request a number in $\{1, 2, \ldots, 20\}$ was picked uniformly at random. There is no locality of reference in this trace and we call it NLT.

2. Highly Localized Trace. For each request we picked the ith most recent request with probability p_i, where p_i was, respectively, $\frac{22}{200}, \frac{26}{200}, \frac{23}{200}, \frac{20}{200}, \frac{19}{200}, \frac{17}{200}, \frac{15}{200}, \frac{10}{200}, \frac{9}{200}, \frac{8}{200}, \frac{5}{200}, \frac{5}{200}, \frac{5}{200}, \frac{4}{200}, \frac{4}{200}, \frac{2}{200}, \frac{2}{200}, \frac{2}{200}, \frac{1}{200}, \frac{1}{200}$. There is a high locality of reference in this trace (the most recently referenced items have a high probability of being referenced again) and we call it HLT.

The maximum costs these traces were $W_{max}(NLT) = 5593$ and $W_{max}(HLT) = 5740$.

Then we ran the optimal algorithm and computed the optimal cost of the two traces and recorded the CPU time for processing each trace. The maximum breadth of the DAG for NLT and HLT was 10007 and 7871 nodes, respectively. Then the modified approximate algorithm was run on both traces with values for N ranging in 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000. For each run, the (suboptimal) computed cost and the CPU time was recorded. Finally, the LRU (Least Recently Used) and LFD algorithms were executed on each trace. The results are shown in Tables 1, 2, 3 and 4, where we have used the following definitions:

$$\textit{deviation from optimal} = \frac{\textit{cost} - \textit{optimal cost}}{\textit{optimal cost}}$$

$$\textit{normalized cpu time} = \frac{\textit{cpu time}}{\textit{cpu time for optimal}}$$

The CPU times were measured using the Unix's `time` command. This method, although not highly accurate, suits our purpose of comparing relative magnitudes. Also all processes were run on a 200 MHz Pentium Pro PC with 64 MB memory, 256 KB L2 cache and running NetBSD, such that the running program was the only active process. While the absolute values of CPU time can change from machine to machine, we ensured that the normalized values were almost invariant.

N	Cost	Deviation from Optimal	CPU Time (Sec)	Normalized CPU Time
2	2306	74.8%	0.36	0.00002
5	1883	42.7%	0.55	0.00003
10	1711	29.7%	0.89	0.00005
20	1554	17.8%	1.80	0.00010
50	1404	6.4%	4.86	0.00026
100	1350	2.3%	11.59	0.00062
200	1327	0.6%	32.35	0.00173
500	1322	0.2%	144.42	0.00773
1000	1319	0%	476.64	0.02551
2000	1319	0%	1763.67	0.09439
5000	1319	0%	10286.81	0.55052
∞	1319	0%	18685.63	1.00000

Table 1: Results of Breadth Limiting Algorithm on Trace NLT

N	Cost	Deviation from Optimal	CPU Time (Sec)	Normalized CPU Time
2	2771	78.5%	0.39	0.00003
5	2242	44.5%	0.55	0.00004
10	1965	26.6%	0.86	0.00006
20	1816	17.0%	1.57	0.00011
50	1613	3.9%	3.97	0.00028
100	1583	1.9%	10.99	0.00078
200	1565	0.8%	29.27	0.00208
500	1552	0%	136.04	0.00967
1000	1552	0%	474.55	0.03374
2000	1552	0%	1830.23	0.13014
5000	1552	0%	10043.61	0.71417
∞	1552	0%	14063.27	1.00000

Table 2: Results of Breadth Limiting Algorithm on Trace HLT

Tables 1 and 2 show a remarkable tradeoff between accuracy and run time that is possible with our approximation algorithm. As N is increased the cost approaches rather abruptly to the optimal value. The bottom row reflects the values for the optimal algorithm (which does not limit the breadth, i.e. $N = \infty$). (The breadth of the DAG when running

	Cost	Deviation from Optimal	CPU Time (Sec)	Normalized CPU Time
Optimal	1319	0%	18685.63	1.000000
LRU	2293	73.8%	0.00	0.000000
LFD	1486	12.7%	0.01	0.000000

Table 3: Results of LRU and LFD Algorithms on Trace NLT

	Cost	Deviation from Optimal	CPU Time (Sec)	Normalized CPU Time
Optimal	1552	0%	14063.27	1.000000
LRU	2653	70.9%	0.00	0.000000
LFD	1742	12.2%	0.01	0.000000

Table 4: Results of LRU and LFD Algorithms on Trace HLT

the optimal algorithm actually hits a maximum of 10007 and 7871 respectively for NLT and HLT.) As shown in the two table, if we set $N = 100$, the algorithm runs 1612 and 1282 times faster on NLT and HLT, respectively, than the optimal while attaining a cost only about 2% higher than the optimal. Surprisingly, the corresponding figures for both traces are close despite the fundamental difference in their models. This is visualized in Fig. 1. Tables 3 and 4 show the results of running LRU and LFD on the NLT and HLT traces, respectively. The first row reports the figures for the optimal algorithm from Tables 1 and 2 for convenience. Both algorithms, although very fast compared to the optimal and approximation algorithms, produce large deviations from optimal. LFD is closer to the optimal than LRU because it looks ahead in the sequence.

6.2. Periodic Omission Method

Another way to trade optimality for speed, is to let the DAG of the optimal method grow without restriction, but periodically omit all nodes except the best (least cost) one. Thus we modify the optimal algorithm by adding the following step between step 3 and 4:

$3'$. *If k mod $T = 0$, then omit all nodes except one with the least cost,* where k is the discrete time, and T is the period of omissions. We picked the same set of items and the same traces NLT and HLT and performed this modified algorithm with parameter T ranging in $2, 5, 10, 20, 50, 100, 200, 500$, and tabulated the results in Tables 5 and 6. Again the same definitions are used for *deviation from optimal* and *normalized cpu time.*

We observe that the same tradeoff appears again (Fig. 2). As period T is increased, the normalized CPU time goes up while the deviation from falls sharply.

Our traces are not in any way unique and the above observations can be repeated with arbitrary traces and items. However, we suspect that it is possible, though not easy, to come up with pathological examples for which both approximation algorithms perform poorly. For this to happen a trace must be deliberately generated such that all globally optimal and nearly optimal paths perform very badly in the beginning (so that they are omitted in

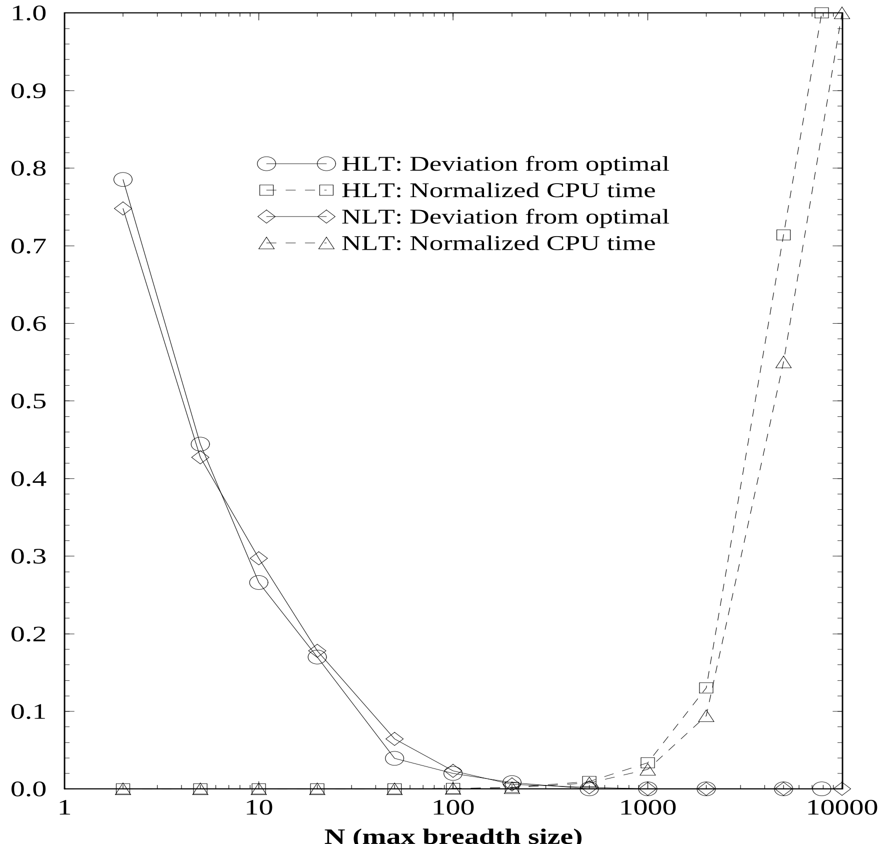

Figure 1: Optimality-Speed Tradeoff by Breadth Limiting

Step $3'$) and the surviving paths in the beginning accumulate a high cost towards the end. However, our numerous experiments show that this is not likely when a trace is generated randomly. Finally, we have no way of prescribing specific values of N or T so that the resulting final cost falls within a given distance of the optimal.

7. LFD is Competitive

Although the approximation algorithms discussed above are orders of magnitude faster than the optimal algorithm, they are yet too slow for majority of interesting applications. Therefore effective heuristics are needed. In Section 4 we showed that LFD is not optimal, now we question whether it is a good heuristic. Competitive analysis, pioneered by Sleator and Tarjan [12], is a well-known way of comparing heuristic solutions. In particular it has been used for on-line algorithms (see for example [3, 6, 10]) and studies whether the cost of an algorithm lies within a constant factor of the optimal cost for every sequence. The constant factor, if any, is then used as a measure for comparing algorithms.

Definition 7.1 *An algorithm is α-*competitive *if its cost is within a constant factor α of optimum on any sequence of requests (up to an additive constant). An algorithm is competitive if it is α-competitive for some constant α. The constant α is called the* competitive

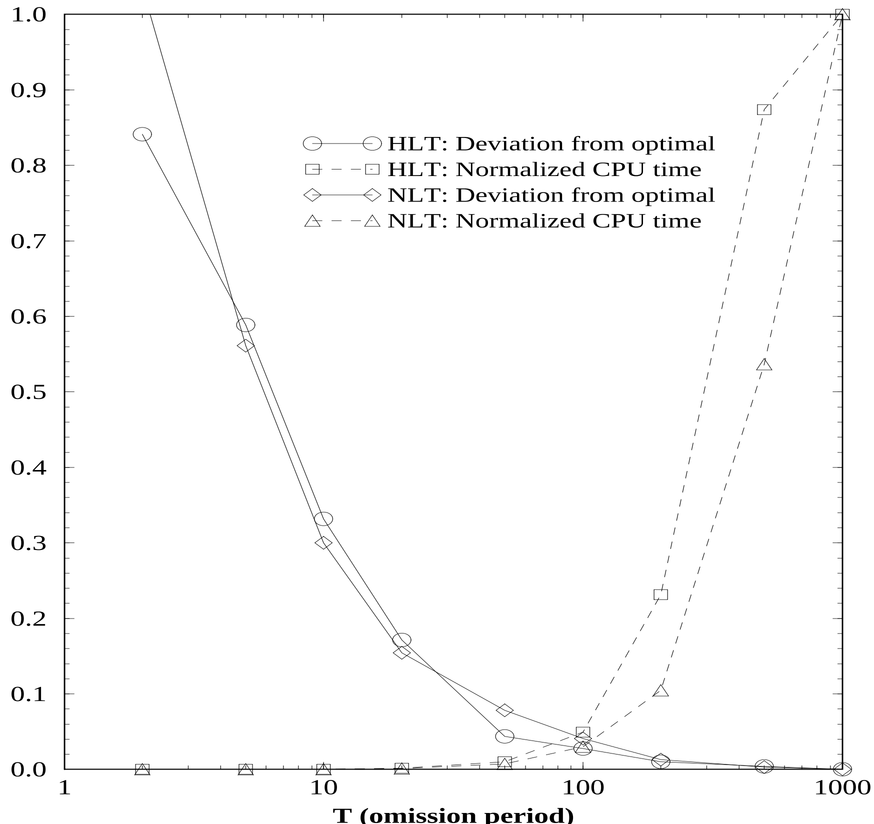

Figure 2: Optimality-Speed Tradeoff by Periodic Omissions

factor. *An algorithm is* strongly competitive *if it achieves the smallest possible competitive factor.*

The following theorem is a competitive analysis of the LFD algorithm in the case of uniform size and nonuniform cost items.

Theorem 7.1 *Given a set* $\mathbb{U}$ *of items with unit size and nonuniform cost* $c_1, c_2, \ldots, c_n$*, and a cache of size* B *in initial state* $\mathcal{B}_0$*, the following holds for all sequences* ρ*:*

$$\frac{W(A_{lfd}, \rho, B)}{W(A_{opt}, \rho, B)} \leq \frac{c_{max}}{c_{min}} \ ,$$

where $c_{min} = \min_{i \in \mathbb{U}}\{c_i\}$ *and* $c_{max} = \max_{i \in \mathbb{U}}\{c_i\}$*.*

Proof. Pick a request sequence ρ. An optimal algorithm A_{opt} on ρ produces a state sequence $\{\mathcal{B}_k^*\}_{k=1}^m$ from which we can determine the sequence $\{\delta_k^*\}_{k=1}^m$. Thus we have:

$$\sum_{k=1}^{m} \delta_k^* \, c_{min} \leq W(A_{opt}, \rho, B) = \sum_{k=1}^{m} \delta_k^* \, c_{\sigma_k} \leq \sum_{k=1}^{m} \delta_k^* \, c_{max} \ .$$

T	Cost	Deviation from Optimal	CPU Time (Sec)	Normalized CPU Time
2	2684	103.5%	0.69	0.00004
5	2059	56.1%	1.36	0.00007
10	1715	30.0%	3.83	0.00020
20	1523	15.5%	18.40	0.00098
50	1422	7.8%	139.67	0.00747
100	1373	4.1%	562.63	0.03011
200	1336	1.3%	1958.36	0.10481
500	1323	0.3%	10026.21	0.53657
1000	1319	0%	18685.63	1.00000

Table 5: Results of Periodic Omission Algorithm on Trace NLT

T	Cost	Deviation from Optimal	CPU Time (Sec)	Normalized CPU Time
2	2858	84.1%	0.51	0.00004
5	2466	58.9%	1.77	0.00013
10	2067	33.2%	6.07	0.00043
20	1818	17.1%	22.65	0.00161
50	1620	4.4%	139.94	0.00995
100	1595	2.8%	693.50	0.04931
200	1568	1.0%	3257.17	0.23161
500	1558	0.4%	12292.84	0.87411
1000	1552	0%	14063.27	1.00000

Table 6: Results of Periodic Omission Algorithm on Trace NLT

On the other hand, LFD on ρ produces possibly another state sequence $\{\mathcal{B}'_k\}_{k=1}^m$ with corresponding $\{\delta'_k\}_{k=1}^m$, and we have:

$$W(A_{lfd}, \rho, B) = \sum_{k=1}^{m} \delta'_k \, c_{\sigma_k} \leq \sum_{k=1}^{m} \delta'_k \, c_{max} \ .$$

Also we have: $W(A_{opt}, \rho, B) \leq W(A_{lfd}, \rho, B)$. Now suppose $c_1 = c_2 = \ldots = c_n = c_{max}$ (i.e. a problem with uniform size and uniform cost items) and notice that the same state sequences $\{\mathcal{B}^*_k\}_{k=1}^m$ and $\{\mathcal{B}'_k\}_{k=1}^m$ are still valid state sequences for ρ. Also notice that the latter state sequence is the result of LFD on ρ for the new problem and hence it is optimal [2]. Therefore: $\sum_{k=1}^m \delta'_k \, c_{max} \leq \sum_{k=1}^m \delta^*_k \, c_{max}$. Hence

$$\sum_{k=1}^{m} \delta^*_k \, c_{min} \leq W(A_{opt}, \rho, B) \leq W(A_{lfd}, \rho, B) \leq \sum_{k=1}^{m} \delta^*_k \, c_{max} \ ,$$

$$q c_{min} \leq W(A_{opt}, \rho, B) \leq W(A_{lfd}, \rho, B) \leq q c_{max} \ ,$$

where $q = \sum_{k=1}^m \delta^*_k$. Therefore

$$W(A_{lfd}, \rho, B) \leq q c_{max} \ ,$$

$$W(A_{opt}, \rho, B) \geq q c_{min}$$

and the desired conclusion follows. □

This result implies that LFD is at least $\frac{c_{max}}{c_{min}}$-competitive. We do not know that, for a set of items with given sizes and costs, whether this factor is the least possible for LFD. However it implies that for all possible sequences its cost is no more than a constant multiple of the optimal. There are policies, e.g. LFU (Least Frequently Used), for which this is not true [12].

8. LRU is Competitive

LRU is a widely-used heuristic that performs well in memory paging. Sleator and Tarjan [12] showed that for uniform size and cost items, LRU is strongly competitive with a competitive factor of B, where B is the size of the cache. The following theorem states that it is competitive in the generalized case.

Theorem 8.1 *Given a set $\mathbb{U}$ of items of size $a_1, a_2, \ldots, a_n$ and of cost $c_1, c_2, \ldots, c_n$, and a cache of size B at initial state $\mathcal{B}_0$, the following holds for all sequences ρ:*

$$W(A_{lru}, \rho, B) \leq \alpha \; W(A_{opt}, \rho, B) + \beta \ ,$$

where α and β are constants independent of ρ.

Proof. Pick a sequence ρ. LRU on ρ produces a sequence of hits and misses indicated by $\{\delta_k\}_{k=1}^m$. Let us divide ρ into phases separated by a sequence of times $t_1, t_2, \ldots$, starting with the first miss time,

$$\rho = \sigma_1, \ldots, \underbrace{\sigma_{t_1}}_{\text{first miss}}, \underbrace{\sigma_{t_1+1}, \ldots, \sigma_{t_2}}_{\text{first phase}}, \underbrace{\sigma_{t_2+1}, \ldots, \sigma_{t_3}}_{\text{second phase}}, \ldots$$

such that

$$t_{i+1} = \min \left\{ j \mid \sum_{k=t_i+1}^{j} \delta_k a_{\sigma_k} \geq B - a_{min} \right\} ,$$

where $a_{min} = \min_{i \in \mathbb{U}} \{a_i\}$. During each phase, the total size of items entering the cache equals or slightly exceeds the cache size. We claim that an optimal algorithm must incur at least one miss on each phase. Consider the two cases that can happen during a phase.

<u>Case 1</u>: LRU misses on distinct items.
We have two sub-cases depending on the last miss (say on i) which happened before the beginning of current phase. (Note: Each phase ends with a miss.)

<u>Case 1a</u>: There is a miss to i in a current phase.

$$\ldots, \underbrace{\sigma_{t_j} = i}_{\text{last miss before current phase}}, \underbrace{\sigma_{t_j+1}, \ldots, \sigma_k = i, \ldots, \sigma_{t_{j+1}}}_{\text{current phase}}, \ldots$$

Notice that at time t_j item i is brought into cache and is later pushed out by some other items only if i becomes the least recently used item in cache. Therefore if there is another miss on i, it means that all other items that are resident in cache at time k came after time t_j but before time k, and their total size added to a_i is equal to or greater than B. This implies that an optimal algorithm must miss at least one of these items which are all in the current phase.

<u>*Case 1b*</u>: There is no miss to i in a current phase.

$$\ldots, \underbrace{\sigma_{t_j} = i}_{\text{last miss before current phase}}, \underbrace{\sigma_{t_j+1}, \ldots, \sigma_{t_{j+1}}}_{\text{current phase}}, \ldots$$

An optimal algorithm at time $t_j + 1$ contains item i and during the current phase more items arrive none of which is i. Since their total size is greater than B, at least one of the new items must be out of cache at time t_j forcing the optimal algorithm to incur a miss in the current phase.

<u>*Case 2*</u>: Two misses happen on the same item i in a current phase.

$$\ldots, \underbrace{\sigma_{t_j+1}, \ldots, \sigma_k = i, \ldots, \sigma_l = i, \ldots, \sigma_{t_{j+1}}}_{\text{current phase}}, \ldots$$

Notice that after the first miss at time t_j item i is brought into the cache and must be pushed out by new items before the second miss at time $t_j + 1$ happens. Therefore, the total size of items that arrived after t_j and before time $t_j + 1$, added to a_i must exceed the cache size B. This forces the optimal algorithm to miss at least one item.

Thus we proved that on every phase of LRU an optimal algorithm misses at least one item. The factor α is found by maximizing the cost incurred by LRU on a phase divided by the cost incurred by an optimal algorithm on the same phase , i.e.

$$\alpha = \max_{\mathcal{G} \subset \mathbb{U}} \left\{ \frac{\sum_{i \in \mathcal{G}} c_i}{c_j} \mid j \in \mathcal{G} \text{ and } \sum_{i \in \mathcal{G}} a_i \leq B \right\} .$$

Now notice that ρ has a start-up period (before the first miss) on which both LRU and an optimal algorithm incur zero cost. Also notice that ρ may terminate with an incomplete phase. On an incomplete phase, an optimal algorithm may incur zero cost while LRU can incur a non-zero finite cost β. In the worst case, β is equal to the maximum cost that an incomplete phase can have, i.e.

$$\beta = \max_{\mathcal{G} \subset \mathbb{U}} \left\{ \sum_{i \in \mathcal{G}} c_i \mid \sum_{i \in \mathcal{G}} a_i \leq B - a_{min} \right\} .$$

Now we can conclude that since every sequence can be divided into a start-up period, a number of complete phases and a final incomplete phase, then the cost of LRU on any sequence is upper-bounded by $\alpha W(A_{\text{opt}}, \rho, B) + \beta$. □

We showed that LRU is α-competitive. However, it remains a question whether α is the least competitive factor for LRU or whether LRU is strongly competitive.

9. Does prefetch help?

Our definition of generalized caching does not leave room for prefetch. In this section we discuss whether there is a benefit in prefetching of items. The answer to this question depends on when costs are actually incurred. If the purpose of caching is merely to minimize retransmissions, then costs are incurred when a transmission occurs—whether due to a miss or because of a prefetch. On the other hand, if the purpose is only to minimize the waiting time of users when requesting items, then costs are incurred when requests are made. (In practice, both goals are desirable and hence the actual cost is a weighted combination of both types of costs.) We show that in the former case, there is no advantage in prefetch.

A *prefetch* at time k is defined as a transition from $\mathcal{B}_k$ to $\mathcal{B}_{k+1}$ such that $\mathcal{B}_{k+1} = (\mathcal{B}_k - \mathcal{E}) \cup \{\sigma_l\}$ where $l > k$, and we assume it occurs only if $\sigma_k \in S$. This type of transition was not allowed in our definition of the problem. Now we prove that if it is allowed, it will not improve performance.

Theorem 9.1 *Given a universe set $\mathbb{U}$, a cache of size B, a request sequence ρ, and a state sequence $\{\mathcal{B}_k\}$ on ρ that contains prefetches, there exists another state sequence $\{\mathcal{B}'_k\}$ that contains no prefetch and its cost is no more than the cost of the former, i.e. $W(\{\mathcal{B}'_k\}, \rho, B) \leq W(\{\mathcal{B}_k\}, \rho, B)$.*

Proof. Set $\mathcal{B}'_0 = \mathcal{B}_0$. Define $\{\mathcal{B}'_k\}_{k=1}^m$ as follows:

1. If $\sigma_k \in \mathcal{B}'_{k-1}$, then set $\mathcal{B}'_k = \mathcal{B}'_{k-1}$.
2. If $\sigma_k \notin \mathcal{B}'_{k-1}$ and $\sigma_k \in \mathcal{B}_{k-1}$, then set $\mathcal{B}'_k = (\mathcal{B}'_{k-1} - \mathcal{E}') \cup \{\sigma_k\}$, where $\mathcal{E}'$ is the same set $\{\mathcal{B}_k\}$ purged when it last prefetched σ_k.
3. If $\sigma_k \notin \mathcal{B}'_{k-1}$ and $\sigma_k \notin \mathcal{B}_{k-1}$, then set $\mathcal{B}'_k = (\mathcal{B}'_{k-1} - \mathcal{E}') \cup \{\sigma_k\}$, where $\mathcal{E}'$ is equal to the same set purged by $\{\mathcal{B}_k\}$ at this time.

Thus $\mathcal{B}'_k$ is determined in the case of all possible events. It is seen that $\{\mathcal{B}'_k\}$ replaces exactly the same items $\{\mathcal{B}_k\}$ does for each request. This implies that corresponding to any item fetched by $\{\mathcal{B}'_k\}$ there is a fetch (or prefetch) of the same item by $\{\mathcal{B}_k\}$. Therefore $W(\{\mathcal{B}'_k\}, \rho, B) \leq W(\{\mathcal{B}_k\}, \rho, B)$. Note that $\{\mathcal{B}'_k\}$ does not prefetch at all. The proof is complete. $\Box$

Corollary 9.1 *For any optimal state sequence $\{\mathcal{B}_k\}$ we can find a state sequence $\{\mathcal{B}'_k\}$ that is optimal and does not prefetch.*

Proof. Let $\{\mathcal{B}'_k\}$ be a state sequence derived from $\{\mathcal{B}_k\}$ according to the rules 1, 2, 3 given in previous theorem. It does not prefetch and the theorem guarantees that $W(\{\mathcal{B}'_k\}, \rho, B) \leq W(\{\mathcal{B}_k\}, \rho, B)$. Since $\{\mathcal{B}_k\}$ is optimal then $W(\{\mathcal{B}_k\}, \rho, B) \leq W(\{\mathcal{B}'_k\}, \rho, B)$. Therefore $\{\mathcal{B}'_k\}$ is optimal. $\Box$

Theorem 9.1 establishes that prefetch has no benefit if a prefetch is as costly as the corresponding fetch. In fact, it is easily seen that prefetch can possibly increase the total cost because either the prefethed item may have to be purged before its anticipated request happens, or it may purge items that might be requested soon.

Corollary 9.1 implies that we can disallow prefetch without degradation in performance.

10. Conclusion

We studied the generalized caching problem and proved that it is NP-complete. It was shown that LFD is not optimal but is competitive. We presented two methods for obtaining nearly optimal answers without expending inordinate amounts of CPU time. It was also shown that LRU remains competitive in the generalized case, and proved that a classical result on ineffectiveness of prefetch extends to this case. There are a number of open problems outstanding. It is not known if the problem remains NP-complete if only either cost or size is non-uniform. We also do not know if LRU is strongly competitive among generalized on-line algorithms. Whether the competitive factor of LFD is the minimum among generalized off-line algorithms remains a question as well.

Acknowledgement. The author expresses gratitude for the helpful comments of Professors Jerome R. Cox, Jr., Subhash Suri and George Varghese of Computer Science Department.

References

[1] V. A. Aho, P. J. Denning, J. D. Ullman, "Principles of Optimal Page Replacement," *Journal of the ACM,* Vol. 18, No. 1, 80–93, January 1971.

[2] L. A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer, *IBM System Journal,* 5 (2) 78–101, 1966.

[3] Allan Borodin, Nathan Linial, Michael E. Saks "An Optimal On-line Algorithm for Metrical Task System," *Journal of the Association for Computing Machinery,* Vol. 39, No. 4, October 1992.

[4] Michael R. Garey, David S. Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness," *W.H. Freeman and Company,* San Francisco, 1979.

[5] Saied Hosseini-Khayat, Jerome R. Cox, Jr., "Optimal Solution of Off-line and Online Generalized Caching," *Technical Report WUCS-96-20,* Department of Computer Science, Washington University.

[6] Mark S. Manasse, Lyle A. McGeoch, Daniel D. Sleator, "Competitive Algorithms for Server Problems," *Journal of Algorithms,* No. 11, pp. 208–230, 1990.

[7] R. L. Mattson, J. Gecsei, D. R. Slutz, I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM System Journal,* 5 (2), 78–117, 1970.

[8] World Wide Web home page of the National Laboratory for Applied Network Research: *http://www.nlanr.net/Cache.*

[9] Christos H. Papadimitriou, S. Ramanathan, P. Venkat Rangan, "Information Caching for Delivery of Personalized Video Programs over Home Entertainment Channels," *Proceedings of The IEEE International Conference on Multimedia Computing and Systems,* Boston, MA, May 1994.

[10] Prabhakar Raghavan, Marc Snir, "Memory Versus Randomization in On-line Algorithms," *Automata, Languages and Programming : Proceedings of the 16th International Colloquium, Stresa Italy, July 1989,* pp. 687–703, Springer-Verlag.

[11] S. Ramanathan, P. Venkat Rangan, "Architectures for Personalized Multimedia," *IEEE Multimedia,* Vol. 1, No. 1, Spring 1994.

[12] D. D. Sleator, R. E. Tarjan, "Amortized Efficiency of List Update and Paging Rules," *Communications of the ACM* Vol. 28, No. 2, pp. 202–208, February 1985.

[13] Allan J. Smith, "Bibliography on Paging and Related Topics," *Operating Systems Review,* vol. 12, 39–56, Oct. 1978.

[14] Allan J. Smith, "Second Bibliography for Cache Memories," *Computer Architecture News,* Vol. 19, No. 4, June 1991.

[15] Thomas Stephenson, Harry Voorhees "IMACTS: An Interactive Multiterabyte Image Archive," 14^{th} *IEEE Symposium on Mass Storage Systems,* 1995.