Evolving En-Route Caching Strategies for the Internet

Jürgen Branke[1], Pablo Funes[2], and Frederik Thiele[1]

[1] Institute AIFB, University of Karlsruhe, 76128 Karlsruhe, Germany
branke@aifb.uni-karlsruhe.de
[2] Icosystem Corp., 10 Fawcett ST. Cambridge MA 02138 USA
pablo@icosystem.com

Abstract. Nowadays, large distributed databases are commonplace. Client applications increasingly rely on accessing objects from multiple remote hosts. The Internet itself is a huge network of computers, sending documents point-to-point by routing packetized data over multiple intermediate relays. As hubs in the network become overutilized, slowdowns and timeouts can disrupt the process. It is thus worth to think about ways to minimize these effects. Caching, i.e. storing replicas of previously-seen objects for later reuse, has the potential for generating large bandwidth savings and in turn a significant decrease in response time.
En-route caching is the concept that all nodes in a network are equipped with a cache, and may opt to keep copies of some documents for future reuse [18]. The rules used for such decisions are called "caching strategies". Designing such strategies is a challenging task, because the different nodes interact, resulting in a complex, dynamic system. In this paper, we use genetic programming to evolve good caching strategies, both for specific networks and network classes. An important result is a new innovative caching strategy that outperforms current state-of-the-art methods.

1 Introduction

The Internet is a distributed, heterogeneous network of servers. Besides other services, it can be regarded as a huge distributed database. Access to documents on the net is mostly based on a strict client-server model: a client computer generates a request, opens up a connection to a server host, and retrieves the document from the server. This naturally creates a lot of network traffic and, in case of congestions, sometimes causes significant response times or *latencies*. Therefore, it is necessary to think about ways to minimize network traffic.

One starting point is the observation that some popular documents are requested all the time, while others are almost never requested. Therefore, it makes sense to store copies (*replicas*) of popular documents at several places in the network. This phenomenon has prompted server companies (e.g., Akamai [1]), to create forms of mirroring to save bandwidth by servicing requests from hosts that are closer, in Internet topology terms, to the clients making the requests.

K. Deb et al. (Eds.): GECCO 2004, LNCS 3103, pp. 434–446, 2004.

However, this solution obviously works only for long-term data access patterns in which a commercial interest can be matched with monetary investments in distributed regions of the globe.

For a broader perspective, observe that when two neighbors on a block request the same document, two independent channels to the remote server hosting the document are created, even though both requesting computers are connected to the same trunk line. The same data is sent over from the server twice, and relayed by a common chain of routers in between. It would make sense, for any of the intermediate hosts, to keep a copy of the document, allowing it to service the second request directly, without having to contact the remote host at all. Clearly, this would result in a dramatic reduction in network traffic and latency.

Proxy servers sometimes have this capability, being able to optimize Internet access for a group of users in a closed environment, such as a corporate office or a campus network. Much better savings and scalability are possible by using this strategy at all levels: If the campus proxy fails to retrieve the page from the cache, or even, if the two requests come from neighboring university campuses in the same city, then a node further down the chain would have the opportunity of utilizing its cache memory, for the same opportunity exists in every single router on the Internet.

The difficult question is to decide which documents to store, and where to store them. With finite memory, it is impossible for individual hosts to cache all the documents they see.

A global policy control in which a centralized decision-making entity distributes replicas among servers optimally is impractical, for several reasons: the tremendous complexity, because the Internet is dynamically changing all the time, and because no global authority exists. Thus, each server has to decide independently which documents it wants to keep a replica of. The rules used for such decisions are also known as *caching strategies*.

Today, many routers with caching — such as a campus network proxy — use the well-known LRU (Least Recently Utilized) strategy: objects are prioritized by the last time they were requested. The document that has not been used for the longest time is the first to be deleted. Although this makes sense for an isolated router, it is easy to see why LRU is not an optimal policy for a *network* of caching hosts. In our example above, all the intermediate hosts between the two neighbors that requested the same document, and the server at the end of the chain, will store a copy of the document because a new document has the highest priority in LRU. However, it would be more efficient if only one, or a few, but not all intermediate nodes kept a copy. In isolation, a caching host tries to store all the documents with highest priority. In a network, a caching host should try to cache only those documents that are not cached by its neighbors.

Designing good caching strategies for a network is a non-trivial task, because it involves trying to create global efficiency by means of local rules. Furthermore, caching decisions at one node influence the optimal caching decisions of the other nodes in the network. The problem of cache similarity of the above example is one of *symmetry breaking*: when neighbors apply identical, local-information based strategies, they are likely to store the same documents in their caches.

In this scenario, the network becomes saturated with replicas of the same few documents, with the consequent degradation of performance.

In this paper, we attempt to design good caching strategies by means of genetic programming (GP). As we will show, GP is able to evolve new innovative caching strategies, outperforming other state-of-the-art caching strategies on a variety of networks.

The paper is structured as follows: first, we will cover related work in Section 2. Then, we describe our GP framework and the integrated network simulator in Section 3. Section 4 goes over the results based on a number of different scenarios. The paper concludes with a summary and some ideas for future work.

2 Related Work

There is a huge amount of literature on caching at the CPU level (see e.g., [17]). Caching on a network, or web caching, has only recently received some attention. For a survey see [20, 7].

Generally, the literature on web caching can be grouped into two categories:

1. Centralized control: this is the idea of a central authority overseeing the entire network, deciding globally which replicas should be stored on which server. Thereby, it is usually assumed that the network structure and a typical request pattern are known.
2. Decentralized control: in this group, decentralized caching strategies are proposed, i.e., rules that allow each server to independently decide which recently seen documents to keep. Since these rules are applied on-line, it is important that they can be processed efficiently and that they do not cause additional communication overhead.

The centralized control version is also known as the "file allocation problem". For a classification of replica placement algorithms, see [10]. In [13], an evolutionary algorithm is used to find a suitable allocation of replicas to servers. Besides retrieval cost, [16, 15] additionally consider the aspect of maintaining consistency in the case of changes to the original document, an aspect which we deliberately ignore in our paper. In any case, while the centralized approach may be valid for small networks, it becomes impracticable for larger networks, as the data analysis, the administrative costs, and the conflict between local authorities take over.

Given the difficulties with a centralized approach, in in this paper we focus on decentralized control. We try to find simple strategies that can be applied independently on a local level, thereby making a global control superfluous. This approach is more or less independent of the network structure and request pattern, and can thus much quicker adapt in a changing environment.

Early work on web caching has simply used or adapted traditional caching strategies from the CPU level, such as *LRU* and *LFU* (Least Frequently Used), see e.g. [21]. The disadvantages of these, namely that they lead to cache symmetry, were described in the introduction. An example of a network-aware caching strategy is *GDSF* (Greedy Dual Size Frequency), which considers the number of

times a particular document has been accessed, its size and the cost to retrieve that document from a remote server (in our case, the distance, measured in hops, to the server holding the next replica).

GDSF's cost-of-retrieval factor avoids the cache repetition problem by reducing the priority of documents that can be found in nearby caches. It has been shown to be among the best caching strategies for networks [6, 9]. Another comparison of several web caching strategies can be found in [2].

An interesting alternative has recently been suggested in [18]. There, a node holding a document and receiving a request uses a dynamic programming based method to determine where on the path to the requesting node replicas should be stored. While this approach certainly holds great potential, it requires that all nodes in the network cooperate, and the computation of the optimal allocation of replicas to servers is time-consuming. Furthermore, request frequency information must be stored not only for documents in the cache, but all documents ever seen.

As has already been noted in the introduction, in this paper we attempt to evolve such a decentralized caching strategy by means of GP. A previous example of using GP for the design of caching strategies was demonstrated by Paterson and Livesey [14] for the case of the instruction cache of a microprocessor. Some of their fundamental ideas are similar to ours: GP is a tool that can be used to explore a space of strategies, or algorithms for caching. However, the nature of the CPU cache is different from the problem of distributed network caching because it does not involve multiple interconnected caches.

3 A Genetic Programming Approach to the Evolution of Caching Strategies

In this section, we will describe the different parts of our approach to evolve caching strategies suitable for networks. We will start with a more precise description of the assumed environment and the network simulator used, followed by the GP implementation.

3.1 Network Simulator

The overall goal of our study was to evolve caching strategies for complex data networks like the Internet. However, for testing purposes, we had to design a simplified model of the network.

Edges and Nodes. Our network consists of a set of servers (nodes), connected through links (edges). Each server has a number of original documents (which are never deleted), some excess storage space that can be used for caching, and a request pattern. We assume that the shortest paths from each server to all other servers as well as the original locations of all documents are known. Thereby, we are obviating the problem of *routing* and focusing only on *caching* decisions.

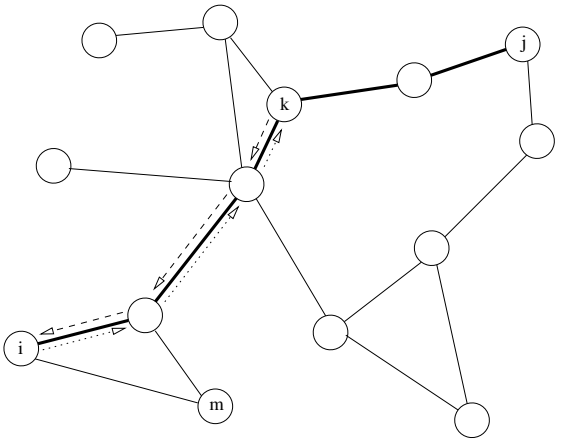


Fig. 1. If node i requests a document from server j, the request is sent along the shortest path (bold), and the first replica found on this path is sent back (say from node k). Other possible replicas not on the path are ignored (e.g., a replica on node m)

When a server requests a document from a remote server, it sends out out this request along the route (shortest path) to the remote server that is known to keep the original copy of the document. All nodes in between check whether they have a cached replica of the requested document. If not, the request is passed through. When either a cached replica or the original document has been found, the document is sent back to the requesting server, following the inverse route. All the intermediate nodes receive the document's packets and pass them along the path to the next node, until the document arrives at its destination. An example is depicted in Figure 1.

Bandwidth, Queuing and Packetizing. When a document travels through the network, it is divided into many small packets. Each link has an associated bandwidth, being able to deliver a limited number of bytes per second. Excess packets wait in an infinite FIFO queue until they can be serviced. We simplified the network operation somewhat by ignoring timeouts.

Efficiency. The main goal is to minimize the average latency of the requests, which is defined as the average time from a request until the arrival of (the last packet of) the document.

3.2 Evolving Caching Strategies

Caching as Deletion. It is easy to see that servers which have not yet filled up their memories, should store every single document they see. Even if an oracle was able to tell that a particular object will never be accessed again, there would

Table 1. Functions used for GP

Function	Meaning	Function	Meaning
add(a,b)	$a + b$	sin(a)	$\sin(a)$
sub(a,b)	$a - b$	cos(a)	$\cos(a)$
mul(a,b)	$a \cdot b$	exp(a)	e^a, $a \in [-100, 100]$
div(a,b)	$\begin{cases} \frac{a}{b} & : \quad (b \neq 0) \\ 1 & : \quad (b = 0) \end{cases}$	iflte(a,b,c,d)	$\begin{cases} c & : \quad (a < b) \\ d & : \quad (a \geq b) \end{cases}$

be no harm in storing it. The problem comes when the caching memory fills up, and a storage action requires the deletion of some other previously cached object. Cache machines have finite disk space and so they must eventually discard old copies to make room for new requests.

In order to evolve caching strategies we thus focused on the problem of deletion. The nodes in our simulated networks store all the documents they receive, until their caches are filled up. If, however, there is not enough memory available to fit the next incoming one, some space must be freed up. All cached objects are sorted according to a priority function, and the document with the lowest priority is trashed. The operation is repeated until enough documents have been deleted so that there is enough space to save the newcomer. In the remainder of this paper we shall define *caching strategy* as the priority rule used for deletion.

Genetic Programming. Given the difficulties to define a restricted search space for caching strategies, we decided to use genetic programming [11, 12], which allows an open-ended search of a space of more or less arbitrary priority functions.

In order to explore the space of caching policies, we employed a generic set of GP functions (Table 1), combined with a set of terminals representing the local information available about each document (Table 2). Each node collects this *observable* information that can be gathered from the objects themselves as they are sent, received and forwarded. That is, for example, why the distance we use is the number of hops a document traveled before reaching the host (observed distance) as opposed to the true distance to the nearest replica, which could only be determined through additional communication.

Another important decision was to avoid measures that need to be recomputed before each usage. This allows a host to maintain its cache sorted rather than re-computing and re-sorting before each deletion. Access frequency, therefore, is calculated as one over the average mean time between accesses, at the time of the last access (see [2]).

Since our focus was more on the application than the search for optimal parameter settings, we used rather standard settings for the test runs reported below: GP has been run with a population size of 60 for 100 generations, the initial population has been generated randomly with depth of at most 6. Tournament selection with tournament size of 2 was used, and a new population was generated in the following way:

Table 2. Terminals used for GP

Variable	Long name	Meaning
A	timeCreated	Time when replica was stored in cache
B	size	Size of document in kilobytes
C	accessCount	Number of times the document has been accessed
D	lastTimeAccessed	Time of last access to document
E	distance	Distance from node that sent the document (in number of hops)
F	frequency	Observed frequency of access (in accesses per second)
$\Re$		Random constant

1/3 of the individuals were simply transferred to the next generation
1/3 of the individuals were generated by crossover
1/3 of the individuals were generated by mutation only.

Crossover was the usual swapping of sub-trees, mutation replaced a sub-tree by a new random tree.

3.3 Evaluating Caching Strategies

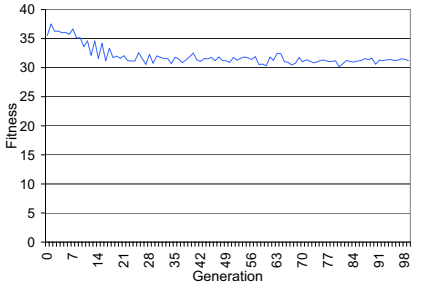
Evaluating caching strategies analytically is difficult. Instead, we tested them, using the simulation environment described in Section 3.1. There are two possible scenarios: if the network topology, and the location and request patterns of all documents are known, GP can be used to tailor a caching strategy exactly to the situation at hand. Evaluation is deterministic, as we can simulate the environment exactly and simply test the performance of a particular caching strategy in that environment. We made preliminary tests with fixed networks and obtained excellent results (not included in this paper due to space restrictions).

On the other hand, such topology and patterns may be known only approximately (e.g., "the document request frequencies follow a scale-free distribution"). We would like to evolve caching strategies that perform well in a whole class of possible scenarios. In other words, we are looking for a solution that applies in general to all networks within the range of characteristics we expect to find in the real world. Since it is impossible to test a caching strategy against all possible networks, we defined classes of scenarios and tested each caching strategy against a random representative from that class. Of course, that made the evaluation function stochastic. Following observations from other evolutionary algorithms used for searching robust solutions [4], we decided to evaluate all individuals within one generation by the same network/request pattern, changing the scenario from generation to generation. Naturally, elite individuals transferred from one generation to the next have to be re-evaluated.

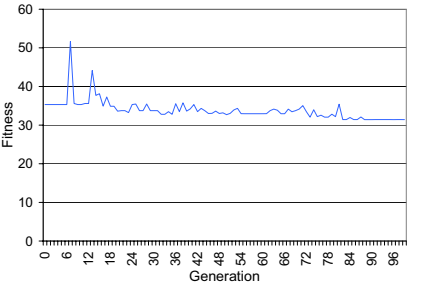
Note that using a simulation for evaluation is rather time consuming. A single simulation run for the larger networks used in our experiments takes about 6 minutes. Thus, even though we used a cluster of 6 Linux workstations with 2 GHz each, and even though we used a relatively small population size and only 100 generations, a typical GP run took about 5 days.

Node	0	1	2	3	4	5	6	7	8	9
Original	0	1	2	3	4	5	6	7	8	9
Replica	5	6	7	8	9	0	1	2	3	4

Fig. 2. Optimal distribution of replicas on a simple linear network



(a) Fitness of best individual per generation



(b) Performance of best individual on 10 other request patterns

Fig. 3. Exemplary GP-run on linear network

4 Results

4.1 Linear Networks

First, we tested our approach by evolving strategies for linear networks, for which we were able to determine the optimal placement of replicas, analytically.

The first test involved a purely linear network with 10 nodes connected in a line (i.e. the first and last node have exactly one neighbor, all other nodes have exactly two neighbors). Every node has one original document, each document has the same size, and each node has excess memory to store exactly one additional document. The request pattern is uniform, i.e. each document is requested equally often (on average). Then, the optimal distribution of replicas is depicted in Figure 2 (for a proof see [19]).

A typical test run is shown in Figure 3, where part (a) shows the observed average latency of the best individual in each generation, as observed by GP, and part (b) shows the performance of these individuals on 10 additional random request patterns to test how the solution generalizes. The caching strategy evolved is rather complex and difficult to understand. Its performance, however, was excellent.

Table 3 compares the latency of different caching strategies over 30 additional tests with different random request patterns. Note that GP is a randomized method, and starting with different seeds is likely to result in different caching

Table 3. Average latency of different caching strategies on the linear network ± standard error.

Caching Strategy	Ø latency
OPTIMAL	31.58 ± 0.03
BESTGP	31.98 ± 0.06
GDSF	47.67 ±1.17
DISTANCE	50.40 ± 1.24
RANDOM	61.65 ± 0.14
LRU	74.77 ± 0.19

strategies. Therefore, we run GP 5 times, and used the average performance over the resulting 5 caching strategies as result for each test case. OPTIMAL is the latency observed with the optimal distribution of replicas on this linear network, which is a lower bound. As can be seen, the strategy evolved by GP (BESTGP) was able to find a solution that performs very close to the lower bound. Other standard caching strategies as GDSF or LRU, perform poorly, LRU even worse than RANDOM,[1] where it is randomly decided whether to keep or delete a particular document. Looking at distance only (DISTANCE) is better than looking at the last time accessed only (LRU), but also much worse than the evolved strategy.

4.2 Scale-Free Networks

The Internet has a scale-free structure, with a few servers being highly connected, and many servers having only few connections [8, 22, 3]. The tests in this subsection are obtained using a scale-free network with 100 nodes. We used the method described in [5] to generate random scale-free networks with similar characteristics as the Internet. There are 100 original documents with document size between 0.1 and 2 MB. In each of 1000 simulated seconds, an average of 1100 requests are generated in a simulated Poisson process. Request frequencies are also scale-free: some documents are requested much more often than others.

With the characteristics of the network defined only as distributions, we searched for individual strategies that could perform well over a wide range of networks in the class. As explained in Section 3.3, this was achieved by using a different random network in each generation.

As it turned out, the observed latencies varied dramatically from one simulation to another, the reason being that large latencies are generated when one or more edges receive more requests than their bandwidth allows them to service. As predicted by queuing theory, a link can be either underutilized (bandwidth $>$ request load) or overutilized (bandwidth $<$ request load). In the first case, latency remains close to zero, and in the second it grows to infinity. We have deliberately set the parameters in our simulation so as to generate networks close

[1] A random strategy, in spite of being completely blind, has the advantage of breaking symmetry (cf. page 435) because all hosts use different random number seeds, thus their caches tend to complement each other

Table 4. Comparison of different caching strategies on the class of scale-free networks with 100 nodes. Table shows average rank according to latency ± standard error.

Caching Strategy	Ø rank(latency)
RUDF	1.13 ± 0.06
GDSF	2.80 ± 0.15
DISTANCE	3.10 ± 0.28
LRU	3.70 ± 0.16
RANDOM	4.27 ± 0.14

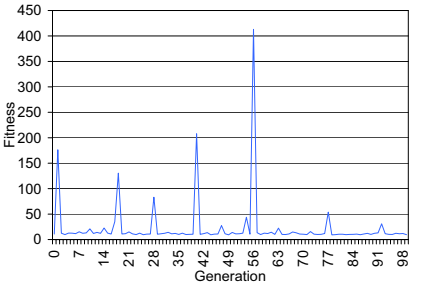
to the saturation point. Depending on the random topology and request pattern, a network can be underutilized, leading to latencies in the order of milliseconds, or overutilized, leading to latencies of of several seconds.

GP had some difficulties with this high variance, which led us to evaluate each individual three times per generation, and take the average as fitness. Even then, the randomness was substantial, as can be seen in the oscillations in performance of the best solution in Figure 4 (again, part (a) shows the fitness of the best individual as observed by GP, while part (b) shows the performance of that individual on 20 additional tests). Nevertheless, the results obtained were convincing. Although GP again generated some rather good but complicated caching strategies, in one run it came up with a surprisingly simple strategy that could be further simplified to the following:

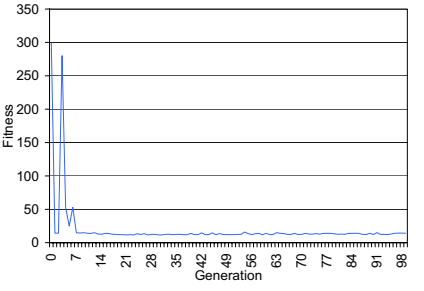
$$priority = lastTimeAccessed \cdot (distance + accessCount)$$

This result is quite astonishing as it adds two totally different parameters, the distance the document has traveled and the number of times it has been accessed. But the combination has meaning: it keeps the documents that have either a very high access count, or a very high distance, but only if they have been accessed recently (otherwise lastTimeAccessed would be small). We call this newly evolved strategy *RUDF* for "Recently Used Distance plus Frequency". This rule performs very well in comparison to all other caching strategies tested (see Table 4), and thus sets a new benchmark. Again, we used 30 test runs to evaluate a strategy, with a new random network and request pattern generated for each test instance. Due to the large variance between test instances, a comparison based on mean latency has only limited explanatory power. We therefore used a non-parametric measure, the average rank the caching strategy achieved when compared to the other strategies tested (with 1 being the best rank, and 5 being the worst rank). The results show that RUDF works well over the randomly generated range of networks in the considered class of scale-free networks.

In order to test how RUDF would perform on completely different networks, we additionally tested it on the linear network from above, and on two scale-free network classes with 30 and 300 nodes, respectively. On the linear network, it resulted in a latency of 35.8 ± 0.42, i.e. worse than the caching strategy evolved particularly for the linear network, but still better than all other tested strategies. The results on the two scale-free network classes are shown in Tables 5 and 6. As



(a) Fitness of best individual per generation



(b) Performance of best individual on 20 other instances

Fig. 4. Exemplary GP-run on the class of scale-free networks

Table 5. Comparison of different caching strategies on the class of scale-free networks with 30 nodes. Table shows average rank according to latency ± standard error.

Caching Strategy	Ø rank(latency)
RUDF	1.03 ± 0.03
GDSF	1.97 ± 0.03
LRU	3.03 ± 0.03
RANDOM	4.20 ± 0.07
DISTANCE	4.70 ± 0.10

Table 6. Comparison of different caching strategies on the class of scale-free networks with 300 nodes. Table shows average rank according to latency ± standard error.

Caching Strategy	Ø rank(latency)
RUDF	1.03 ± 0.03
DISTANCE	2.23 ± 0.16
GDSF	3.20 ± 0.11
RANDOM	4.20 ± 0.16
LRU	4.33 ± 0.13

can be seen, RUDF significantly outperforms all other strategies independent of the network size. DISTANCE seems to be very dependent on the network size, it performs second on large networks, but worse than RANDOM on smaller ones.

5 Conclusions and Future Work

A key inefficiency of the Internet is the tendency to retransmit a single blob of data millions of times over identical trunk routes. Web caches are an attempt to reduce this waste by storing replicas of recently accessed documents at suitable locations. Caching reduces network traffic as well as experienced latency.

The challenge is to design caching strategies which, when applied locally in every network router, exhibit a good performance from a global point of view. One of the appeals of GP is that it can be used to explore a space of algorithms. Here, we have used GP to search for caching strategies in networks that resemble the Internet, with the aim to find strategies that minimize latency. A new rule called RUDF was evolved, which is very simple yet outperformed all other tested caching strategies on the scenarios examined.

An important obstacle we faced was measuring fitness, because fitness could only be determined indirectly through simulation, and different random seeds resulted in a high variance in latency (the criterion we used as fitness). Nevertheless, averaging and multiple-seed evaluation techniques allowed us to evolve robust strategies that are efficient in a wide variety of conditions.

There are ample opportunities to extend our research: first of all, it would be necessary to test the newly evolved caching strategy on a larger variety of networks. Then, the caching strategies could be made dependent on node characteristics (e.g., location in network, number of connections), moving away from the assumption that all nodes should apply identical caching strategies. Finally, we could have independent GPs running on every node, so that the caching strategies on the different nodes would coevolve.

References

1. http://www.akamai.com/en/html/services/edgesuite.html.
2. H. Bahn, S. H. Noh, S. L. Min, and K. Koh. Efficient replacement of nonuniform objects in web caches. *IEEE Computer*, 35(6):65–73, 2002.
3. A.L. Barabasi, R. Albert, and H. Heong. Scale-free characteristics of random networks: the topology of the world-wide web,. *Physica A*, 281:2115, 2000.
4. J. Branke. Reducing the sampling variance when searching for robust solutions. In L. Spector et al., editor, *Genetic and Evolutionary Computation Conference (GECCO'01)*, pages 235–242. Morgan Kaufmann, 2001.
5. T. Bu and D. Towsley. On distinguishin between internet power law topology generators. Technical report, Depatrment of Computer Science, University of Massachusetts, 2002.
6. L. Cherkasova and G. Ciardo. Role of aging, frequency, and size in web cache replacement policies. In B. Hertzberger, A. Hoekstra, and R. Williams, editors, *High-Performance Computing and Networking*, volume 2110 of *LNCS*, pages 114–123. Springer, 2001.
7. B. D. Davison. A web caching primer. *IEEE Internet Computing*, 5(4):38–45, 2001.
8. M. Faloutsos, P. Faloutsos, and C. Faloutsos. On power-law relationships of the internet topology. In *Proceedings of the ACM SIGCOMM, Sept 1999.*, 1999.
9. S. Jin and A. Bestavros. Greedydual* web caching algorithm. *Computer Communications*, 24(2):174–183, 2001.
10. M. Karlsson, C. Karamanolis, and M. Mahalingam. A framework for evaluating replica placement algorithms. Technical Report HPL-2002-219, Hewlett-Packard, 2002.
11. J. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* MIT Press, Cambridge, 1992.
12. J. Koza. *Genetic Programming II: Automatic Discovery of Reusable Programs.* MIT Press, Cambridge, 1994.
13. T. Loukopoulos and I. Ahmad. Static and adaptive data replication algorithms for fast information access in large distributed systems. In *International Conference on Distributed Computing Systems*, pages 385–392, 2000.
14. N. Paterson and M. Livesey. Evolving caching algorithms in C by GP. In *Genetic Programming: Proceedings of the Second Annual Conference.* Morgan Kaufmann, 1997.

15. G. Pierre, M. Van Teen, and A. Tanenbaum. Dynamically selecting optimal distribution strategies for web documents. *IEEE Transactions on Computers*, 51(6):637–651, 2002.
16. S. Sen. File placement over a network using simulated annealing. *ACM*, 1994.
17. A. J. Smith. Cache memories. *ACM Computing Surveys*, 14(3):473–530, 1982.
18. X. Tang and S. T. Chanson. Coordinated en-route web caching. *IEEE Transactions on Computers*, 51(6):595–607, 2002.
19. F. Thiele. Evolutionäre Optimierung von Caching Strategien für das Internet. Master's thesis, Institute AIFB, University of Karlsruhe, 76128 Karlsruhe, Germany, 2004.
20. J. Wang. A survey of web caching schemes for the internet. *ACM SIGCOMM Computer Comm. Rev.*, 29(5):36–46, 2001.
21. S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, and E. A. Fox. Removal policies in network caches for World-Wide Web documents. In *Procedings of the ACM SIGCOMM '96 Conference*, Stanford University, CA, 1996.
22. S. Yook, H. Jeong, and A. Barabasi. Modeling the internet s large-scale topology. PNAS October 15, 2002 vol. 99 no. 21, 2002.