

The Multicache Simulation Environment

Version 1.0 Reference Manual

Ismail Ari
ari@cs.ucsc.edu

UCSC-CRL-03-02
June 2, 2003

Department of Computer Science
University of California, Santa Cruz

ABSTRACT

`Multicache` is a trace-driven cache simulator developed to make the design, analysis, and comparison of cache replacement algorithms in multi-level caching systems practical. To make the simulator generic and flexible architectural specifics (such as those found in processor caches) or protocol (HTTP, TCP/IP) details were excluded. The simulator is best for quick implementation, analysis and comparison of new cache replacement algorithms that deal with web-proxy objects, file system files, disk blocks, memory pages, and other fixed or variable size objects.

The simulator is easy to learn and expand. It provides many building blocks for well-known replacement algorithms (LRU, LFU), space management routines, container data structures, and simple topologies. Topologies like client-server, server-storage pairs, web proxy hierarchies, and peer-to-peer networks can be implemented.

Modern caching related concepts such as using heterogeneous algorithms, filtering effects, demotions are also evaluated. The structure of the simulator allows multiple algorithms to have control over one cache space, allowing adaptive techniques to be tested. The code is open source and is written in C++.

Keywords: Simulation, multi-level caches, web hierarchy, distributed, adaptive, topology generator, filtering, heterogeneous caching.

Contents

1. Introduction	4
1.1 What Multicache does	4
1.2 What Multicache doesn't do	5
1.3 Related Simulators	5
1.3.1 Web cache simulators	5
1.3.2 Network Simulator (NS)	5
1.3.3 Disk simulators (DiskSim, Pantheon)	5
1.3.4 Processor cache simulators	5
2. Running Multicache	6
3. Simulator Design	7
3.1 Fixed Configuration	8
3.2 N-Level Linear Caches	9
3.2.1 Exclusive caching	10
3.2.2 Demotions improve exclusive caching	10
3.2.3 Using heterogeneous policies improves exclusive caching	10
3.3 Topology Configuration	11
3.4 Cache Dependency	11
3.5 Requests	13
3.6 Hash With Chaining	14
3.7 Event Scheduler	14
4. Algorithms	15
4.1 Optimal (OPT) Replacement	15
4.2 Random Replacement	16
4.3 Recency-Based: LRU, GCLOCK, 2NDCHANCE	16
4.4 Sequential Workloads: MRU	16
4.5 FIFO, LIFO	16
4.6 Popularity, Frequency of Access: LFU, MFU	17
4.7 SIZE	17
4.8 Hybrids: GDS, GDSF, LFUDA, GD	17
4.9 Multi-List Algorithms	17
4.9.1 LRU-K	17
4.9.2 2Q, MQ	18
4.10 Adaptive Algorithms	18
4.10.1 TwoExpertFixed	18
4.10.2 TwoExpertAdaptive	18
4.11 Dynamic Space, "Tug-Of-War"	19
4.11.1 Adaptive Replacement Cache (ARC)	19
4.11.2 ACME: Vote-based	19
4.11.3 Loss Updates Using Machine Learning	19

4.12	Predator	20
4.13	Other Metrics and Algorithms	20
5.	Topologies	21
5.1	Routing and Peer Discovery	21
6.	Conclusions	23
7.	Appendix	24
7.1	Installation	24
7.2	Statistics Package	24
7.3	Write Operations and Cache Consistency	24
	References	25

List of Figures

1.1	Topologies seen in various distributed systems	4
2.1	A sample trace file. Each line consists of the time, id, and size fields, respectively. The ID space was preprocessed to make offsets globally unique in this trace. Note that size is in multiples of block size, which is 1024 bytes.	6
3.1	Simulator Design. Multicache program is the driver for the simulator. It creates the algorithms and the configurations in which these algorithms are used.	7
3.2	Pseudo-code for the get-request function that defines the general operation for each cache configuration type.	8
3.3	Conversion of block range requests to page requests in block-level configuration.	9
3.4	Pseudo-code for conversion from variable-size block ranges to fixed-size pages.	9
3.5	A simple N level cache. The object request of client resulted in a <i>hit</i> in the second level and was responded locally.	10
3.6	This figure illustrates the set-based methodology for finding the correlation between three replacement algorithms. Objects in area 7 are cached by all policies at the given time. Areas 1, 2, 4 denote the objects cached by only one policies and areas 3, 5, 6 denote objects cached by two policies. The list on the left shows the hit/miss results based on where the object was located when it was requested.	13
3.7	Hash with chaining: The objects with ids that hash to the same value are chained to each other.	14
4.1	The figure on the left was given in Cormen <i>et al.</i> 's [10] for the illustration of a minimal set-cover problem. In this case sets S3, S4, and S5 are enough to cover all the points. Similarly, the hits achieved by an optimal algorithm may be achieved by a set of cache replacement algorithms. OPT expertise is obtained as a combination of the expertise of weaker experts.	15
5.1	A sample WAN topology generated by the Tiers program.	21
5.2	Floyd's algorithm for finding shortest path in complex topologies.	22

1. Introduction

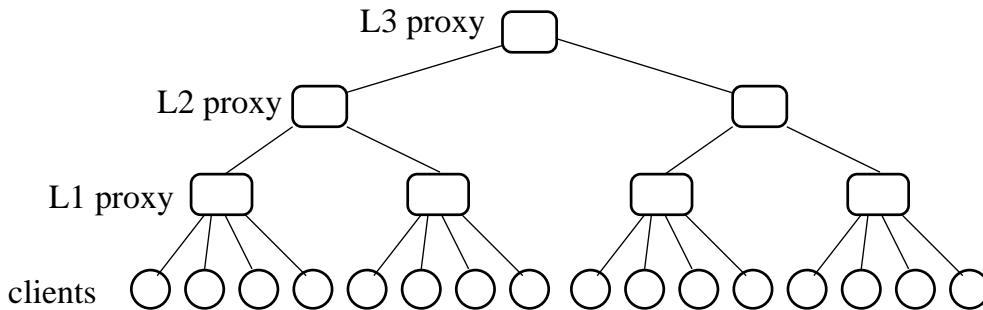
The concept of “multi-level caching” includes a wide range of topologies such as client-server pairs [34], linearly connected “en-route” caches [28], distributed file systems [17], hierarchical or collaborative web proxies [8] (Fig. 1.1(a) and 1.1(b)), and global peer-to-peer systems [22]. Our goal is to design a simulator that not only measures the performance of a cache in isolation, but also as a part of a global caching system.

In this paper the words “algorithm” and “policy” are used interchangeably to mean *cache replacement algorithms*. Note that the word “algorithm” points to the mechanisms involved, where “policy” emphasizes decisions and expertise.

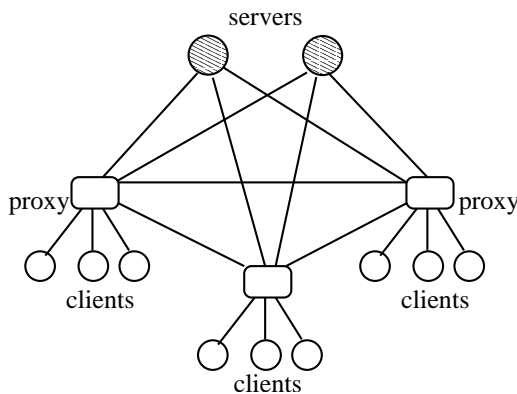
1.1 What Multicache does

Multicache does:

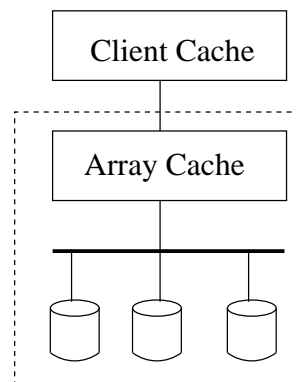
- Run traces that are either synthetically generated or collected from real systems
- Implement cache replacement algorithms in detail
- Simulate messaging by calls to `get-request()` and `get-response()` functions
- Enable two-level, N-level, tree-like and other hierarchical and distributed cache topologies
- Include a minimal routing capability based on shortest paths (under development).



(a) Simplified topology for hierarchical web proxies



(b) Collaborative proxies



(c) A simplified storage system model [34]

Figure 1.1: Topologies seen in various distributed systems

1.2 What Multicache doesn't do

Multicache does not:

- Run real system applications and track the accesses to real physical memory
- Implement finite state machines of HTTP, and TCP/IP protocols
- Open real sockets and connections
- Packetize requests or send protocol specific control messages
- Employ complex routing algorithms (yet).

The items excluded would limit our simulator to be useful for simulation of only one type of system. However, we want to test distributed caching with workloads seen in the web, file systems, by block devices, databases and various other enterprise systems¹. We avoid system specific details whenever possible.

1.3 Related Simulators

To the best of our knowledge there is no other *freely available, multi-level* cache simulator that can make different *policies interact the way* our simulator can in *one or more* cache nodes.

1.3.1 Web cache simulators

All the other multi-level cache simulators that we are aware of are hierarchical web proxy simulators modeled after the Squid such as the “Wisconsin web cache simulator”, DavisSim, Proxycizer, NCS and NS (links to all can be found at <http://www.web-cache.com/simulators.html>). These are geared towards analyzing the web workload and have specific focuses on HTTP, TCP, DNS protocols and packet-level issues.

1.3.2 Network Simulator (NS)

NS [12] which is the de-facto simulator for networking research. It has modules for web proxy caching in addition to the detailed models of transfer protocols (transport, network, MAC) and physical channels. The focus in NS *web modules* is on HTTP request and response packets and again networking issues. We are less concerned about the transport, network, MAC protocol or physical channel details.

1.3.3 Disk simulators (DiskSim, Pantheon)

DiskSim has validated modules that simulate disks, intermediate controllers, buses, device drivers, request schedulers, disk block caches and disk array data organizations. It is the de-facto open source storage sub-system simulator. Pantheon is the industry counter-part of DiskSim that is developed at HP Labs. Both of these simulators have block cache management routines.

1.3.4 Processor cache simulators

Processor cache simulators [16] are focused on the effects of associativity, cache lines sizes, TLB, and other processor specific details. Dinero IV [11] is a trace driven uniprocessor cache simulator for memory reference.

¹If these details are critical to the reader, either other simulators described in Section 1.3 should be considered or multicache should be extended with the required capabilities

2. Running Multicache

```
Usage:multicache <trace file name>
Using the sample trace.
===== Policy List =====
[0]RAND [1]LRU [2]MRU [3]LFU [4]MFU [5]FIFO
[6]LIFO [7]SIZE [8]GDS [9]GDSF [10]LFUDA
[11]GD [12]LRUK [13] GCLOCK [14] 2NDCHANCE
[15]2Q [16]TWOEXPERT [17] OPT

===== Simulation Configurations =====
[0] Fixed Configuration
[1] N-Level Fixed Configuration
[2] Topology Configuration
[3] Cache Dependency Test
Select configuration type:
```

The current version of the simulator has a non-graphical, interactive Application Program Interface (API). When the user runs `multicache` from the command prompt the **usage** information and the initial menu is displayed. The user has to input the name of a **trace file** as an argument. The sample trace shown in Figure 2.1 is a block storage system trace. Each line (or record) in the trace file is simply formatted as `<time, id, size>`. For block level traces `id` is the offset in a given disk and `size` is the number of bytes accessed after the given offset. If no trace file is specified the simulation starts with a default sample trace.

A **policy list** indicates the currently available algorithms that have already been implemented. Next, the possible simulation **configurations** are listed and the user is asked to select one. The `main()` function for the `multicache` simulator can be found in `multicache.cc`.

The next section describes the structure of the simulator, the design choices and the specific configurations listed in the initial menu.

29.490601 60001019752 8192
29.811752 40000133679 1024
30.228225 60000586216 8192
30.279827 60000586264 32768
30.303053 60000586376 16384
30.317305 60000586216 8192
30.329732 60000586296 16384
30.355997 60000586352 24576
30.880745 40000960532 4096
30.941857 10000001832 8192
30.978885 10000002272 8192

Figure 2.1: A sample trace file. Each line consists of the time, id, and size fields, respectively. The ID space was preprocessed to make offsets globally unique in this trace. Note that size is in multiples of block size, which is 1024 bytes.

3. Simulator Design

Figure 3.1 shows the general framework of our simulator. Multicache is the main program to construct the simulation scenario and coordinate the operation. Initially, a *configuration* which defines the rules between the policies is selected from the *configuration pool* and is created. Configurations can consist of one or more levels of cache with one or more policies interacting in each cache. Then, the policies are selected from the *policy pool* and are initialized. Next, the policies are registered with the cache configuration. Once the configuration is completed, the trace file is opened and parsed line by line. Policies are evaluated and the hit ratios as well as other statistics are recorded into the log files.

Each configuration has a set of algorithms and a topology to play the caching scenarios. The configuration represents the caching system and receives `get_request()` object requests from the clients. All other cache configurations inherit from a generic class `Conf` and they have to implement a `get_request()` method to receive the requests sent by the clients in the multicache program (`multicache.cc`). Multicache uses the `Conf` API to register algorithms, register topologies, make object requests, allocate cache space, or poll the collected statistics.

Figure 3.2 gives the pseudo-code for the basic structure of a caching operation. Each registered policy is asked whether it has the requested object in its cache. If the requested object is cached by the policy we update the statistics for the object, otherwise we check whether there is enough space to place the object in to the cache. If the object is bigger than the current space of the policy then we do not attempt to allocate space for it, otherwise we keep replacing objects from the cache until we get enough space for the object to be placed.

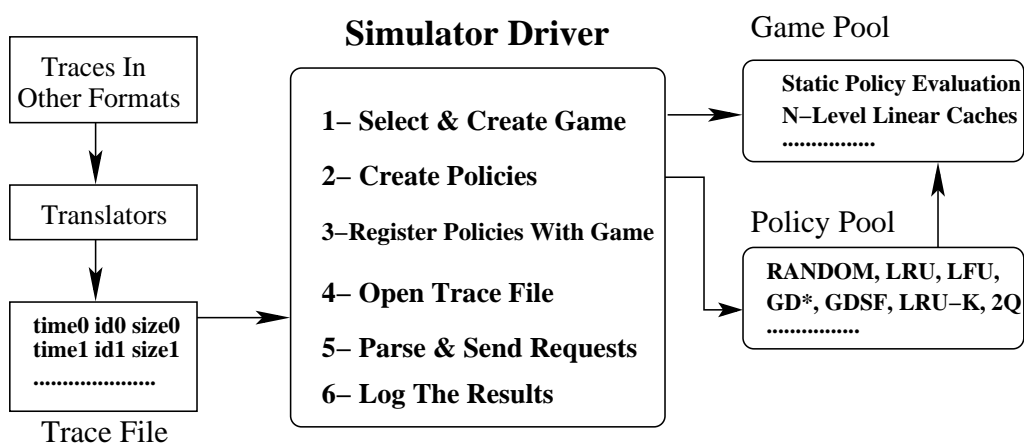


Figure 3.1: Simulator Design. Multicache program is the driver for the simulator. It creates the algorithms and the configurations in which these algorithms are used.

```

FUNCTION GET-REQUEST()
  foreach request
    id ← document requested;
    if Size(id) > CACHE_SIZE then skip request
    foreach policy
      if id ∈ policy cache then
        incrHit(id)
      else
        while insufficient space for id in cache do
          evicted = Replace()
          Cache(id);
        endfor
    endfor
  endfor

```

Figure 3.2: Pseudo-code for the get-request function that defines the general operation for each cache configuration type.

3.1 Fixed Configuration

```

(Continued from the previous menu)
Select configuration type: 0
Select policies separated by space:0 1
Select cache size (in MB):32
Is this page/block level?[Y/N]:n
Do you want to record filtered workload?[Y/N]:n
STARTING SIMULATION!!!
RANDOM 2432 LRU 2616 req=16000

```

Fixed configuration denotes a single policy governing a fixed cache space. It is used to compare various policies under different workloads. Many policies can be tested at the same time, but they will be evaluated separately. Once the fixed configuration is selected the user will be asked to input a few more basic parameters before the experiment begins.

In the example above the user has selected two policies, RANDOM (0) and LRU (1), to be compared and assigned 32 Mbytes of cache space to each. The user inputs “N” or “n” to test variable-size caching (e.g. for web objects). The user may also choose to test fixed-size page replacement using a block level workload (Block-Level Configuration) by saying “Y” or “y” to the block level question. If the filtering is enabled the missed requests of each listed policy will be recorded in a file called `filtered.load.x`, where x denotes the selection order of the policy that was used for filtering. The filtered workload is smaller in size and could be used to test file server, or storage controller policies. In this example, the user has disabled filtering. The simulation started immediately showing the number of requests that were parsed and the number of hits achieved by each policy.

Block configuration calculates the page boundaries, page offset and the number of pages to retrieve for the requested block region as shown in Figure 3.3. Each block region is mapped on to as many as fixed-size memory pages. Figure 3.4 gives the simple pseudo-code for this mapping operation.

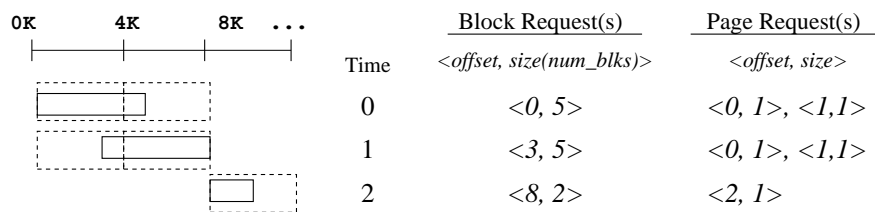


Figure 3.3: Conversion of block range requests to page requests in block-level configuration.

```

PAGE_SIZE = 4096 bytes

start = id/PAGE_SIZE;
len = size/PAGE_SIZE;
if (size < PAGE_SIZE) then len = 1;
foreach policy
  for page = start → < start + len - 1
    policy → Cache(page);

```

Figure 3.4: Pseudo-code for conversion from variable-size block ranges to fixed-size pages.

3.2 N-Level Linear Caches

N-level caches as shown in Figure 3.5 simulate a path of network devices with embedded caching capabilities also known as “en-route caches” [28] or Storage Embedded Networks (SEN). For $N = 2$ it is equal to a client-server model. This capability is achieved by a recursive call to the `get-request()` method of a configuration. The nodes at levels $N \geq 2$ would see a version of the workload filtered by the client caches [30]. This simple enhancement to the fixed configuration made it easier to study the multi-level filtering effects.

```

Select configuration type: 1
How many cache levels will you use:3
Select cache size (in MB):64
Select policy for level 0:1
Select policy for level 1:3
Select policy for level 2:9
Demotions between levels? [0/1]: 1
STARTING SIMULATION!!!
LRU 1189 req=7000

```

In N-level configuration the user first specifies the number of linearly connected caches that are between the clients and the servers. Then, for each level a cache size and a governing policy is specified. In the example above, the user has selected LRU, LFU and GDSF policies to test the benefits of heterogeneous caching in multi-level caches. Next, the demotion mechanism is enabled. A demotion is to transfer a replaced object to the parent cache instead of discarding it. The details of demotions will be discussed later. The final results of the simulation can be found in the `test.out.x.y` output files, where x denotes the node number and y is either “hr” or “bhr” denoting hit rate and byte hit rate, respectively.

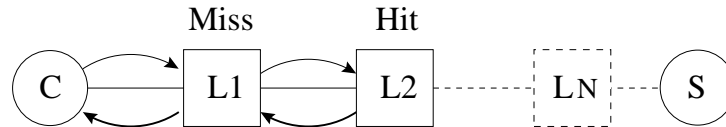


Figure 3.5: A simple N level cache. The object request of client resulted in a *hit* in the second level and was responded locally.

3.2.1 Exclusive caching

In multi-level (linear) caching, if caches are of the same size and if they hold exactly the same elements then a miss in one of them will also result a miss in the other ones. This is called *inclusive caching* and makes upper levels useless. This is usually the case when the same cache replacement policy is used at all levels. The goal is to achieve as much *exclusive caching* [34] as possible between the collaborating caches, so that the cache cluster has the effect of a one big unified cache to the users. Keep in mind that the goal is to improve user perceived performance, but not to provide exclusive caching alone. So, the inherent assumption made with exclusive caching is that the each cache in the multi-level cache system can be accessed much faster than the server or the *source* of the requested object.

3.2.2 Demotions improve exclusive caching

Demote operation moves ejected objects one more hop away from the client (to the parent cache in Figure 1.1(a) and to the array cache in Figure 1.1(c)) instead of discarding them, thus resulting in different objects to be cached in different, but topologically close, caches. However, note that demotions cause extra network overhead and are feasible in LAN or Storage Area Networks (SANs) with high-speed connections. So, some important questions to ask and answer about demotions are:

- How much network overhead do demotions cause?[34] We know that the *demotion rate* is proportional to the miss rate, since every missed object replaces others that get demoted.
- Can *unsolicited* demoted or promoted objects be accepted by all nodes in a cache hierarchy?
- Should statistical information collected for an object be carried to other nodes when this object is demoted? This is similar to *in-band signaling* of information.
- How to handle replicated objects that get demoted from different clients with different statistics?

3.2.3 Using heterogeneous policies improves exclusive caching

Using heterogeneous policies has also been demonstrated to improve exclusivity in multi-level caches [34, 3, 5]. *Promotions*, inverse of demotions, are used for pushing cached objects closer to the clients or edges where they are required most. Optimal placement of objects and replicas of the same object via promotions have been extensively studied [21].

3.3 Topology Configuration

```

Select configuration type: 2
Enter topology file name:topo_ucsc
0 1 1 100 16 1
1 5 1 100 16 1
2 5 1 100 16 1
3 5 1 100 16 1
4 5 1 100 16 1
5 6 1 100 16 1
6 1000 10 10 16 1
Select policy 0 for node 0:1
Select policy 0 for node 1:2
Select policy 0 for node 2:3
Select policy 0 for node 3:4
Select policy 0 for node 4:5
Select policy 0 for node 5:6
Select policy 0 for node 6:7
Correlation (Overlap)=> None [0], Complete [1]:0
STARTING SIMULATION!!!

```

The user first specifies the topology file to be used for the experiment, which in this case is the simplified network topology of University of California Santa Cruz. All of the seven nodes (routers) are assumed have caching capability as would be in Storage Embedded Networks (SEN) [3]. Client Response Time (CRT) measurements are also supported in this configuration. The contents of the topology file (nodes 0 → 6) were printed for demonstration here. Each row of this topology file consists of the following fields:

```
<from node, to node, delay (ms), BW (Mbps), cache size (MB), num. of policies>
```

With this topology file it is possible to create hierarchical tree topologies (many to one), where the next node is known without the need for a routing algorithm. Many-to-many topologies with simple (e.g. shortest path) and complex (Pastry, Brocade, Tapestry) routing schemes is under development. After the policies are selected for each node the clients are synthetically distributed to the nodes in the topology. Their requests could be correlated or uncorrelated. The correlation mechanism currently available in the simulator is explained in a previous paper [5]. The client requests enter into the cache system from the edge nodes.

3.4 Cache Dependency

New cache replacement policies are continuously being designed and the old ones are being tuned for specific workloads. Heterogeneous policies are being used together either to complement each other within the cache hierarchies or to manage the cache resources in a single operating system in a cooperative fashion. It is crucial to understand whether we are really consulting independent cache experts or a group of dependent experts whose overall decision is only as good as one of the experts within the group.

The cache dependency test uses two techniques to quantify the similarities or differences (correlations) between policies. The first technique uses a statistical correlation formula called the Pearson's Product-Moment Correlation Coefficient and the second technique is a set-based heuristic that counts the common objects in multiple policies (as defined by the intersection areas of object sets).

```

[3] Cache Dependency
Select configuration type: 3
Select policy 0:0
Select cache size (in MB):64
Secy policy 1:1
Select cache size (in MB):64
Select policy 2:3
Select cache size (in MB):64
STARTING SIMULATION!!!
RANDOM 2110 LRU 2165 LFU 2109 req=11000

```

In this example the dependency test is selected, and three policies, RANDOM, LRU, and LFU are selected for comparison. The cache sizes governed by each policy are specified as 64 MBytes. Then, the simulation starts immediately printing the hits of each policy.

The first correlation measurement technique is based on statistical procedures that are used to quantify the *relation* among a set of random variables X_1, X_2, \dots, X_n . In the caching case the hits or misses of replacement policies within a time period could be used as the random variables and the relation will be the degree to which these variables vary together or *covary*.

The formula for correlation is driven from the formula for *covariance*. The covariance of two random variables X and Y is a way of measuring the dependency between X and Y . If paired X and Y values tend to both be above or below their means at the same time, this will lead to a high positive covariance. Covariance is defined by:

$$\begin{aligned} cov[X, Y] &= E[(X - E[X])(Y - E[Y])] = \\ &E[X \times Y] - E[X] \times E[Y] = \overline{XY} - \bar{X} \times \bar{Y} \end{aligned} \quad (3.1)$$

Note that if $X = Y$, then the covariance is equal to the variance of the random variable:

$$cov[X, X] = \overline{X^2} - \bar{X}^2 = var(X) = \sigma_X^2. \quad (3.2)$$

and if X and Y are statistically independent, then $E[X \times Y] = E[X] \times E[Y]$ and $cov[X, X] = 0$. The covariance itself must be standardized before it can be useful, since it is sensitive to the standard deviation of X and Y . The Pearson Product-Moment Correlation Coefficient, r , is a simple way to provide this standardization:

$$r = Cov(X, Y) / S_X \times S_Y = SP_{XY} / \sqrt{(SS_X \times SS_Y)} \quad (3.3)$$

where,

$$\begin{aligned} SS_X &= \sigma(X - \bar{X})^2 = \sigma_X^2 - (\sigma_X)^2 / N, \\ SS_Y &= \sigma(Y - \bar{Y})^2 = \sigma_Y^2 - (\sigma_Y)^2 / N, \end{aligned} \quad (3.4)$$

There is a more workable version of the formulas above that was used by the simulator:

$$SS_X = \sum X^2 - \sum X \times \sum X / N \quad (3.5)$$

$$SS_Y = \sum Y^2 - \sum Y \times \sum Y / N \quad (3.6)$$

$$SP_{XY} = \sum X.Y - \sum X \times \sum Y / N \quad (3.7)$$

$$r_{XY} = SP_{XY} / \sqrt{SS_X \times SS_Y} \quad (3.8)$$

$$\rho_{XY} = \sqrt{(1 - (1 - r_{XY}^2)) \times (N - 1) / (N - 2)} \quad (3.9)$$

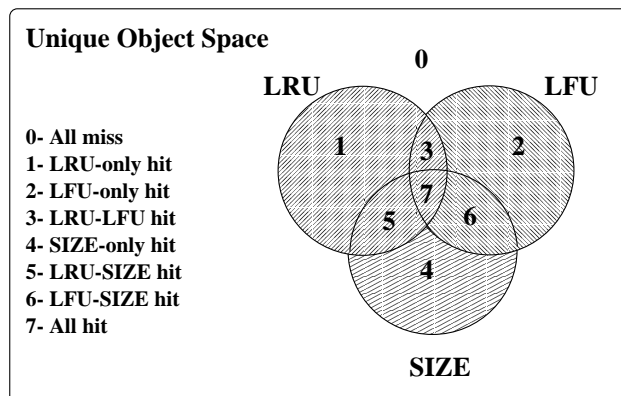


Figure 3.6: This figure illustrates the set-based methodology for finding the correlation between three replacement algorithms. Objects in area 7 are cached by all policies at the given time. Areas 1, 2, 4 denote the objects cached by only one policies and areas 3, 5, 6 denote objects cached by two policies. The list on the left shows the hit/miss results based on where the object was located when it was requested.

Correlations range from -1 (perfect negative relation) through 0 (no relation) and to +1 (perfect positive relation). For a pair of random numbers to be uncorrelated the two streams need to vary randomly around their means with respect to each other.

Figure 3.6 shows the set-based comparison of different policies. Each set denotes the group of objects cached by a certain replacement algorithm at a given time. Note that during warm up all algorithms cache the same objects and then diverge from each other when different object replacement are made. For an infinite amount of cache where no replacement is ever needed, all algorithms cache the same objects and only area 7 exists. Intersection of the sets denote the objects that are cached by two policies. The outer of the union of all sets ($A \cup B \cup C$) consists of the objects that were seen in the past, but that are not currently cached by any algorithm. Objects start at area 7 and move towards the outer areas as they get replaced by algorithms, finally reaching area 0.

3.5 Requests

Objects are created using the given $\langle time, id, size \rangle$ information and are placed in container *nodes* that carry multiple hooks to be iterated by various data structures. For example, algorithms can use a hash with chaining to store the nodes as shown in Figure 3.7. Each node knows its neighboring objects both in the hash and in the list.

`class Node` is a container for requests. It encapsulates a request, also called a *job* and its priority. The Node is list-able and hash-able. Every node has a `prev` and a `next` pointer that point to the neighboring node in the list and the hash. The `hash_next` and `hash_prev` pointers point to the nodes that have the same hash value, since the $HASH(key)$ clashed. The `list_prev` and `list_next` pointers are attached to the neighbors in the order that the requests have been received. List pointers can be used iterate through all the objects and therefore provide a global view.

Currently, there are three types of jobs. The basic one contains the $\langle time, id, size \rangle$ information. The frequency based job additionally has frequency information. A method called `JobFactory` creates a new job that will carry the object information. Algorithms that use specialized types of jobs will implement `JobFactory` differently, but will return a generic `Job*` pointer so that their job could be cached, accessed, retrieved, replaced just like others. The algorithms that have specialized

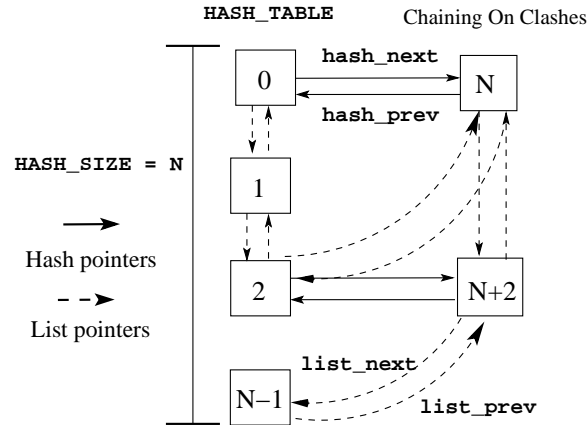


Figure 3.7: Hash with chaining: The objects with ids that hash to the same value are chained to each other.

jobs also need to `dynamic_cast<>` to the proper type to access the detailed statistics associated with that job type.

3.6 Hash With Chaining

Class `Hash` (`hash.cc`, `hash.h`) implements a chained hash. We hash the id using a simple modulus (`HASH_SIZE`) for now. Currently, the `HASH_SIZE` is 4096, though we are also testing 16 KB, 64KB hashes. The calendar queues used in the scheduler of the simulator can resize the hash. If there is a clash of the hash value the new object will be enqueued to the beginning of the list of object that hash to the same value. Class `Hash` contains a `Node**` type `hash_table` and a `Node*` type `first` node. The table will hold pointers to the first node that gets enqueued in each hash location with $O(1)$ access. All access operations for reads and updates are on average $O(1)$. Since, the global linking is done with a list, the `getMin/Max` operations require a very costly $O(n)$ search right now. This limitation of the simulator will be eliminated as a first priority. `Hash` creates the `Nodes` internally for each accepted `Job`. `Enqueue`, `Dequeue`, `adjustPriority`, `getMin/Max`, `incr/decrPriority`, `getJobAtPosition` are some of the methods in the `Hash` API. Total number of objects and total bytes are also recorded for statistics purposes.

`EnqueueJob` method creates a container `Node` and makes the proper pointer attachments both for hash and list. `DequeueJob` accesses the object in $O(1) * \text{numobjects}/\text{HASH_SIZE}$ and then cleans the list and hash pointers. `AdjustPriority` find the object and updates its priority. Again a $O(1) * \text{numobjects}/\text{HASH_SIZE}$ operation. Other functions perform similar access and retrieval functionality.

3.7 Event Scheduler

An event scheduler has been adopted from the NS simulator [12]. It is a priority queue implemented as a “calendar queue”, which resembles the hash structure described above. Calendar queue has capabilities to readjust the hash size and width, so that accesses remain to be $O(1)$.

4. Algorithms

Cache replacement algorithms (`algo.h`, `algo.cc`) provide a priority ordering of objects based on different criteria. Each algorithm is allocated a fixed space to manage and keeps the list of objects that it controls. The total size of the objects should be smaller than or equal to the cache size controlled by this algorithm.

All policies in the simulator inherit from a general class, called `class Algorithm`, and they need to implement certain methods such as `Cache()` and `Replace()`. This polymorphic structure allows algorithms to be easily tested together in one loop. The main simulation triggers specific algorithms using the generic structure given in Figure 3.2.

4.1 Optimal (OPT) Replacement

Belady’s OPT [27] is the theoretical best, off-line cache replacement algorithm. Every online algorithm aims to simulate the behavior of OPT. However, understanding the behavior of OPT is tied to understanding the behavior of different workloads. Unfortunately, workload characterization is extremely complex. Workloads are generated by complex enterprise applications, they mix with each other, and get filtered by caches of multiple levels. Research from the past few decades has found the metrics of recency (inter arrival times, last reference, last-K references), frequency, and size to be major factors affecting the success of cache replacement algorithms.

OPT algorithm replaces the object that will be requested the farthest in the future or “it replaces the page that will not be used for the longest period of time” [27]. To simulate this off-line algorithm together with the other online algorithms (which makes it convenient when plotting the results), we preprocess the trace file and provide a “cheat sheet” for the optimal algorithm. At each request the optimal looks at its cheat sheet to peek into the next time an object will be referenced in the future (*i.e.* an oracle capability). It uses the next reference information as the priority and replaces the object with the highest value.

Theoretically, we classify the cache replacement problem as a “set-cover problem” with respect to the OPT as shown in Figure 4.1. Set-cover problem generalizes from the vertex-cover problem and is NP-hard [10]. We know that no algorithm can hit the objects that OPT misses and the ultimate goal is to hit the same objects that OPT hits. So, ultimately all online cache replacement algorithms try to make an “optimal prediction”.

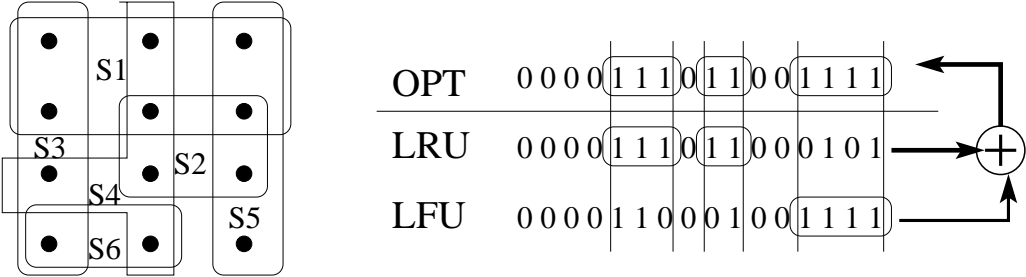


Figure 4.1: The figure on the left was given in Cormen *et al.*'s [10] for the illustration of a minimal set-cover problem. In this case sets S3, S4, and S5 are enough to cover all the points. Similarly, the hits achieved by an optimal algorithm may be achieved by a set of cache replacement algorithms. OPT expertise is obtained as a combination of the expertise of weaker experts.

4.2 Random Replacement

Random gives the base case performance result for cache replacement, since it replaces objects randomly without keeping any access state. In the unbiased random replacement a number between $[0, \text{LISTLEN}]^1$ is randomly selected and the object in the selected list location is replaced. We are testing frequency-biased and size-biased random replacement policies. In the size-biased replacement a byte is randomly selected from the $[0, \text{TOTAL_BYTES}]$ and the object that contains the selected byte is replaced. In the frequency-biased list a point in $[0, \text{TOTAL_1}/\text{FREQUENCY}]$ is selected and replaced. Time bias does not integrate well into these equations, but can be added in the form of dynamic aging similar to the GDSF algorithm described later in this section.

4.3 Recency-Based: LRU, GCLOCK, 2NDCHANCE

Least Recently Used (LRU) is the classic replacement algorithm that exploits the temporal locality of reference. Unfortunately, few systems provide hardware support for a true LRU replacement. GCLOCK and 2NDCHANCE [27] are two approximation algorithms for LRU².

GCLOCK initializes a parameter called `INIT_COUNT` to 2. Both GCLOCK and 2NDCHANCE scan their lists in order and therefore need to remember where they last stopped scanning (`stopnode`). If there is no record of a `stopnode`, then the scan starts with the first node. Priorities of objects are decremented upon scan. Scanning will go on until a job with priority zero is found and replaced. Enough replacements have to be done before the incoming object can be placed in the cache.

2NDCHANCE (the enhanced second-chance in the OS book [27]) uses two bits, `REF_BIT` and `HISTORY_BIT`, to decide on objects to be replaced. 2NDCHANCE places the objects to the queue with an unset bit and sets this bit if the object is accessed. When it scans the list for replacement, (1) if the reference bit is set, it is reset, and the history bit is set; (2) If the history bit is set it is reset; (3) If both bits are reset the object is replaced.

4.4 Sequential Workloads: MRU

MRU replaces the most recently used objects. Although this seems to violate the temporal locality principle it is very well-suited for sequential workloads (audio, video, disk scans) where the cached objects will never be requested again. MRU provides “scan resistance”, which means it does not flush the useful cache content for objects referenced only once.

4.5 FIFO, LIFO

First-In-First-Out (FIFO) and Last-In-First-Out (LIFO) policies replace the objects suggested by their names and do not require any information about the objects to be replaced. FIFO assumes that the order of the requests also defines the worthiness of the objects. FIFO is a pipe and LIFO is a stack.

¹LISTLEN: Number of objects in the list.

²Details can be found on pages 309-312 of OS book [27] by Silberschatz and Galvin

4.6 Popularity, Frequency of Access: LFU, MFU

LFU replaces the least frequently used (unpopular) objects making the assumption that the popular objects will be requested again with a higher popularity. Most Frequently Used (MFU) algorithm assumes that if an object is currently popular it will lose its popularity very soon and thus replaces the most popular objects.

4.7 SIZE

SIZE prefers hit rates (that are reflected as server CPU offloads) over byte hit rates (network offloads). From another point of view, SIZE algorithm assumes that bigger objects will not be requested again and chooses to replace these objects first. SIZE may also be assigned a threshold to replace the objects only above a certain size. SIZE-with-threshold can then be used to implement exclusive caching in multi-level hierarchies [32].

4.8 Hybrids: GDS, GDSF, LFUDA, GD

Greedy-Dual-Size (GDS) [18, 7] replaces the object with the smallest key $K_i = C_i/S_i + L$, where C_i is the retrieval cost of the object, S_i is the size. L is a running age factor that is set to the key value of the objects that are replaced from the cache. GDS with Frequency (GDSF) [4] adds the frequency of access, F_i , into the same equation and replaces the object with the smallest key $K_i = (C_i \times F_i)/S_i + L$. LFU with Dynamic Aging (LFUDA) replaces the object with minimum $K_i = (C_i \times F_i) + L$ [4]. GreedyDual* [18] is a generalization of GDS that captures both long-term popularity and short-term temporal correlations.

4.9 Multi-List Algorithms

We are currently working on implementations of LRU-K [25, 31], 2Q [19], MQ [36] and Unified Buffer Management (UBM) [20] policies.

4.9.1 LRU-K

LRU-K replaces the objects with the smallest penultimate (last K, last 2 for LRU-2) reference. Requests are classified as correlated or uncorrelated based on whether they had been seen more than once within the correlated reference period. Based on the last reference time, objects eligible for replacement³ are detected, and the object with the maximum backward reference (minimum history K or history2 value, if K=2) is replaced. A special LRU-KJob keeps record of the lastRef and History1, History2 parameters for LRU-2.

LRU-2 is a specific implementation of LRU-K algorithm which considers last K accesses to an object to make a replacement decision. It was found by the designers themselves that considering last 2 references was good enough.

LRU-K also keeps a ghost cache, also called the *History list*. History is a meta-data record of an object without the data. Upon access, if a history record exists for the object it is updated. Otherwise, a history entry is created for first time. Many of these history records can be kept for a long time (RETAINED_INFO_PERIOD). A daemon process will clear the HISTORY data structure

³those with lastRef > CORR_REF_PERIOD.

(hash) periodically every *RETAINED_INFO_PERIOD*. This daemon process runs asynchronously in LRU-K and therefore needs to be implemented with a `policySpecific()` routine. The event scheduler will soon replace this function, therefore we skip the details.

The two parameters *RETAINED_INFO_PERIOD* and *CORR_REF_PERIOD* need to be tuned carefully for successful operation. Adaptive Replacement Cache (ARC) algorithm avoids these magic numbers.

4.9.2 2Q, MQ

2Q is an implementation of LRU-K with multiple queues. 2Q uses 3 lists, *A_{in}* managed as FIFO, *A_{out}* managed as LRU, and *Am* again managed as LRU. This implementation is said to have better time bounds. 2Q has a `reclaimfor()` routine also uses two magic numbers. MQ extends 2Q and uses Q0, Q1, and Qout.

4.10 Adaptive Algorithms

4.10.1 TwoExpertFixed

Two-expert fixed algorithm splits the cache space at a magic ratio (assigning more space to the “better” algorithm) and gives the control of each fixed cache to the selected fixed algorithms. Since there are high correlations between the objects cached by many algorithms the duplication is avoided by ordering the algorithms with respect to the sequence they will receive the objects. The missed object is placed into the first list and if it gets accessed (hit) while it is in the first list it is moved to the second list which is controlled by the second algorithm. There are good and bad pairs of algorithms. Even if algorithms are ordered based on their success with a workload, the two best will not necessarily be the best couple due to possible high correlations (in terms of the objects they hold on to and replace) between the two algorithms. What is needed is an “exclusive and useful” expertise. For example MRU may keep many unique objects not selected by other algorithms, but these objects may never be useful. LFU may keep somewhat popular objects that were once also detected by LRU, but later forgotten by LRU and then got requested again (LFU hits, LRU misses).

4.10.2 TwoExpertAdaptive

Two-expert adaptive algorithm embeds the expertise of two different algorithms as virtual (ghost) caches. In this implementation the ghost caches are as big as the physical cache and they both track the space changes in the real cache. Only one algorithm governs the real cache at one time and this the “current best” algorithm as determined by the hit rate success within a certain past window. `CurrBest` decides which object to cache and replace. There is no prefetching in this algorithm. We have found that a pseudo-magic number such as (1000, 10000) will be good enough to provide this simple adaptivity. This number is pseudo-magic, because we know it has to be small for improved responsiveness, but large for stable switching and low CPU costs. These two trade-offs determine the magic value.

Every some number of requests we check the hit rates and pass the control of the cache to the better algorithm. “Being the more successful expert for more than five periods out of ten periods” is the current simple rule to select the winner. `SetHash` is used by adaptive algorithms to pass the control of objects from one expert to the other. We benefit from using the second expert, when the first expert misses but the second one hits. Proper internal size adjustments to the free space are made after object movements between the lists. There is no dynamic cache space management in this algorithm.

4.11 Dynamic Space, “Tug-Of-War”

One intuitive choice for designing adaptive caches is to divide the cache space into fixed size partitions and let a few successful work on separate partitions as seen in the “virtual cache management” by Arlitt, *et al.* [4]. Objects evicted from one partition go to the next until they are moved out of the cache. Their results [4] and our results [2] show that the performance is bound by the performance of the best partition.

Using our simulator we tried the same partitioned setup, but allowed the policies to fight for cache space in a similar fashion to the “tug-of-war” or rope pulling games. The strength of a policy would be proportional to its hit rates. Unfortunately, we found that if one of the policies was good for enough time it could quickly starve the other policies of cache space making it impossible for these policies to regain control of the cache. This would cause monopoly in the cache violating the rules for providing adaptivity under changing conditions.

4.11.1 Adaptive Replacement Cache (ARC)

ARC has four LRU lists: Top(T1, T2), and Bottom (B1, B2). The algorithm simulates recency and frequency at the same time and manages to shift towards the currently best algorithm (out of the two). The algorithm is very detailed, so the reader is encouraged to read the related publication [24].

4.11.2 ACME: Vote-based

ACME assigns weights to objects, keeps the objects with overall highest weighted-vote from all experts and replaces the smallest. Weights are again assigned to experts and are updated. Upon a request, (1) first the real cache is checked whether it has the job or not, (2) if it doesn’t immediately the other level (or server) is contacted to retrieve the object. (3) After the request is handled the virtual caches (VC) are informed and updated. (4) The outcome of VC are compared against the real outcome and virtual policies may incur a loss depending on the difference $Loss = (PhysicalOutcome - VirtualOutcome)$ [3]. (5) Finally, a loss update is done to readjust the weights.

4.11.3 Loss Updates Using Machine Learning

Machine-learning-based adaptive caching schemes [3] can avoid starvation and adapt to changes in real-time and under dynamic conditions. In adaptive caching with machine learning, existing cache replacement algorithms are treated as experts with initially equal weights. As the requests are made by the clients and as the workload proceeds, the weights of experts are automatically changed by the machine learning algorithms based on their recent success on selected metrics such as the *hit rate* or the *byte hit rate*. Each adaptive cache node can tune itself based on the workload it observes and is therefore an autonomous entity. No cache databases or synchronization messages are exchanged between the nodes.

The *Master Algorithm* [14] predicts with a weighted average (y_t') of the experts’ predictions:

$$y_t' = x_t \cdot w_t \quad (4.1)$$

The weights of the policies are generally initialized equally as $w_i = 1/N$, where N is the number of experts. Depending on the true outcome (y_t), which in caching case is hits and misses, the experts incur *loss*. For example a simple loss function may be: $Loss(y_t', y_t) = (y_t' - y_t)^2$, called the *square loss*. Then this loss is used to update the weights. Many forms of *loss update* have been proposed in

the literature [23, 29]. The *Vovk Update* [29] given below is a generalized version of the *Weighted-Majority* algorithm by Littlestone and Warmuth [23].

$$w_{t+1,i} = w_{t,i} \frac{e^{-\eta \times \text{Loss}_i}}{\sum_{i=1}^N w_{t+1,i}} \quad \text{for } i = 1, \dots, N \quad (4.2)$$

where the parameter η is called the *learning rate* and the summation in the denominator provides normalization of weights between $[0,1]$. The weights at time t are multiplied with the exponential factor to obtain the new weights to be used at time $t + 1$. Because of the exponential factor in the formula it is claimed that the loss updates learn too fast, but do not recover fast enough [15]. Therefore, with loss updates the weights of many experts can quickly become zero or very close to zero. Left with very little trust or cache space these inferior algorithms are never given a chance again to prove their success in the future.

Share algorithms [15] try to ensure that weights do not quickly become zero, so that an inferior policy can recover its lost weight if it starts performing well. In the Share algorithm each policy is forced to contribute a loss-proportional part of their weight into a pool:

$$\text{pool} = \sum_{i=1}^n \left(1 - (1 - \alpha)^{\text{Loss}_i} \right) w_{t+1,i}. \quad (4.3)$$

where α denotes the *sharing rate*. After this sharing, the pool is redistributed by giving equal shares to all policies:

$$w_{t+1,i} = (1 - \alpha)w_{t+1,i} + \frac{1}{N-1}(\text{pool} - \alpha w_{t+1,i}) \quad \text{for } i = 1, \dots, N \quad (4.4)$$

The machine-learning-based adaptive techniques have one real cache and virtual caches as many as the experts. A master policy selects the best expert. Weights are assigned to the experts and updated dynamically. Algorithms incur loss when they miss objects and all weights are updated using loss and share updates.

4.12 Predator

Predator is a mediator between the experts and implements dynamic space allocation. Predator controls the weights of the policies in the cache. Predator decides who gives up space and who caches which objects. The decisions are random, but probabilistically biased towards successful policies (`getBest()`) for caching and biased towards policies holding large cache spaces for replacement (`getBiggest()`). This cache configuration is inspired from the the species' fight for fitness in nature in the existence of a predator. `DistributeUtility(distribute.cc, .h)` provides a probabilistic (biased-random) selection for the policies.

4.13 Other Metrics and Algorithms

Usage of retrieval costs [35, 26], objects ID hashes [13], hop counts [3], and Quality of Service (QoS) values [9] have also been proposed for making cache placement and replacement decisions. Unfortunately, we cannot cover all the existing algorithms; rather, our goal is to show that the possible caching criteria and the ways to use them are endless, therefore we need a flexible simulator design for integrating new criteria, policies and schemes.

5. Topologies

Currently, some of the configurations use hard-coded constants for link delays between the caches to calculate an approximate measure of user perceived latency. Although we are less concerned about the transport, network, MAC protocol or physical channel details, we cannot ignore the effects of networks altogether. Therefore, we use a light-weight network generator called *Tiers* [6, 28] to produce static, but realistic, topologies with calculated delays on links (Figure 5.1). Topologies include WAN, MAN, LAN modeling with redundant connections. These topologies are then employed in simulations to measure the overall user perceived latency with acceptable proximity. Leaving some of the networking issues out improves the scalability of the simulations.

Topology consists of a list of edges and array of nodes (vertices). It also has records of cache capacity, routing and caching times. Links are represented by $\langle \text{from}, \text{to}, \text{delay}, \text{BW} \rangle$. There is currently an array of vertices and a single edge for each vertex attaching it to its neighbor. We need a vertex list to allow alternate routes and redundant connections.

5.1 Routing and Peer Discovery

Currently a `getnextto()` function returns the next node from an array, but more involved methods ranging from shortest paths to P2P calculations, will replace this simple method. For calculating shortest paths in large topologies generated by *Tiers* we use Floyd's algorithm.

Dijkstra's algorithm runs in $O(m \log n)$ time, where n is the number of nodes (---V---) and m is the maximum of the number of nodes (---V---) and the number of arcs (---E---). If we want to find minimum distances between all pairs of nodes we could run Dijkstra's algorithm n times with a

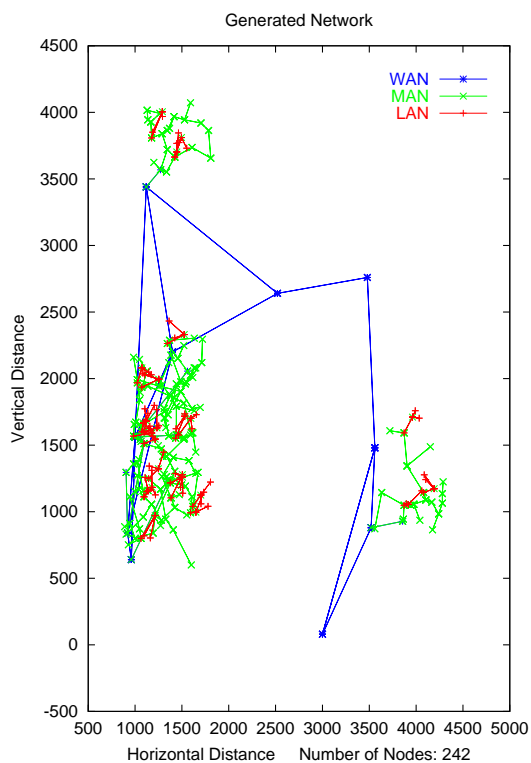


Figure 5.1: A sample WAN topology generated by the *Tiers* program.

```

for (int z = 1; z ≤ MAXNODES; ++z)
  for (int x = 1; x ≤ MAXNODES; ++x)
    for (int y = 1; y ≤ MAXNODES; ++y)
      if (dist[x][z]+dist[z][y] < dist[x][y])
        dist[x][y] = dist[x][z]+dist[z][y];

```

Figure 5.2: Floyd's algorithm for finding shortest path in complex topologies.

different node as the source each time. This would take $O(nm \log n)$ time. Floyd's algorithm is an $O(n^3)$ algorithm, which is clearly worse than Dijkstra's algorithm in many cases, but not too much worse in some cases. In any case, Floyd's algorithm operates on an adjacency matrix and is easier to understand than Dijkstra's algorithm. We will use this algorithm once at the beginning and will probably have less than thousand nodes.

Floyd's algorithm considers each node in turn as a pivot node. When a pivot node z is being considered we check each pair of nodes x, y to see if there is a path from x to y through the pivot z . If there is, and if that path is shorter than the current distance saved for x, y then the length of that path is saved as the shortest distance so far between x and y . Essentially, if we can get from x to z and from z to y then we can get from x to y .

In RTT calculation we differentiate between the forward-going control messages (assumed to be 40 bytes) and returned data messages. Control messages incur the same propagational delay with data. We assume that the bandwidths and delays are symmetric on the forward and backward paths.

6. Conclusions

We designed a cache simulator that makes the analysis of multi-level caching practical. We have implemented a broad set of policies in our policy pool. With this we intend to show that the time to create a new policy and compare it with others has been reduced. We cannot cover all policies, therefore our goal is to provide the base that will allow easy integrations, expansions and interactions of policies for analysis of novel multi-level caching strategies.

Due to the immense amount of research in the caching arena, new caching strategies are inevitably inspired by the previous work and usually come as minor modifications to some of the existing policies. These strategies usually add and track a few more parameters, while leaving the general framework unchanged. This is a motivation to build simulators like ours. Inventors can use basic object-oriented techniques to extend previous algorithms, reuse the matching functionality and overwrite the policy specific ones (place, replace, update) with their own versions without modifying the written code. Today, most researchers in the fields write their own cache simulators possibly duplicating the efforts and causing minor variations in their implementations for complex algorithms that lead to differences in results. We hope to provide a common, efficient and flexible base for the development and analysis of novel cache replacement policies and distributed cache management strategies.

It has been shown in prior work [33, 5, 3] that using different cache replacement policies at different levels of a cache hierarchy improves performance. Different policies could be used together to manage one cache space or used separately in different levels of a hierarchy to complement each other for improved overall hit ratios. However, finding the best combinations of policies and strategies is tedious in today's complex systems [1] without the practical analysis tools in hand. Our simulator helped us with the design and analysis of our own machine-learning-based caching strategies [3] to select the best current policies and adapt to the changes in workloads and network topologies.

Acknowledgements

Ismail Ari was supported by a grant from Hewlett-Packard Laboratories. We thank Dick Henze, Rich Elder, and Scott Marovich in the Storage Technologies Department of Hewlett-Packard Laboratories for supporting our research and providing helpful comments. We are also grateful to members of the Storage Systems Research Center, Manfred Warmuth and the members of the Machine Learning Group at University of California, Santa Cruz for helping us in our ongoing research. Also thanks for anonymous reviewers of conferences WDAS2002, Freenix2003, and HotOS2003.

7. Appendix

7.1 Installation

A Makefile has been provided with the source code and the simulator has been successfully compiled on both Linux2.4 (and some previous versions) and Sun Solaris. Just type `make` and run the application `multicache` after compilation. The API for the simulator is text-based in this version. We are considering a TCL/TK graphical interface.

7.2 Statistics Package

Under development is a statistics package (`class Statistics` and `Correlation`) modeled after `Lintel/Stats.H` in Pantheon disk simulation package developed by HP Labs. Currently, the statistics code in our simulator is woven into the other code. Our goal is to have these capabilities flexible enough to be used for statistical analysis of any random variable throughout the simulator. Capabilities include `max()`, `min()`, `mean()`, `variance()`, `stddev()`, `conf95()`, `relconf95()` and `total()`, `total_sq()`, `covariance()`, `correlation_coeff()`, `is_correlated95()`, `lsq_slope()`.

7.3 Write Operations and Cache Consistency

In OS memory management and file system page cache management a dirty bit is used to specify the modified pages. These (victim) pages should be written back into the disk before they are replaced from the cache. Web proxies use If-Modified-Since (IMS) messages or timeouts to invalidate stale cached copies. Distributed file systems and databases use various mechanisms such as locks and leases to assure consistency. Since our simulator aims to be generic at this time we did not implement cache consistency routines, which are different for web, file systems and databases.

References

- [1] E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch. Hippodrome: running circles around storage administration. In *Proceedings of the 2002 Conference on File and Storage Technologies (FAST)*, Monterey, CA, Jan. 2002.
- [2] I. Ari. Storage Embedded Networks (SEN) and Adaptive Caching Using Multiple Experts (ACME). Proposal For Advancement To Ph.D. Candidacy, University of California, Santa Cruz, CA, 2002.
- [3] I. Ari, A. Amer, R. Gramacy, E. L. Miller, S. A. Brandt, and D. D. E. Long. ACME: adaptive caching using multiple experts. In *Distributed Data and Structures*, volume 4. Carleton Scientific, 2002. Extended version of the WDAS 2002 workshop paper.
- [4] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin. Evaluating content management techniques for web proxy caches. In *Proceedings of the 2nd Workshop on Internet Server Performance (WISP '99)*, Atlanta, Georgia, May 1999.
- [5] M. Busari and C. Williamson. Simulation evaluation of a heterogeneous web proxy caching hierarchy. In *Proceedings of the 9th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS '01)*, pages 379–388, Cincinnati, OH, Aug. 2001. IEEE.
- [6] K. L. Calvert, M. B. Doar, and E. W. Zegura. Modeling internet topology. *IEEE Communications Magazine*, 35(6):160–163, 1997.
- [7] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS '97)*, pages 193–206, Dec. 1997.
- [8] A. Chankhunthod, P. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell. A hierarchical Internet object cache. In *Proceedings of the 1996 USENIX Annual Technical Conference*, San Diego, CA, 1996.
- [9] J. Chuang and M. Sirbu. Stor-serv: Adding quality-of-service to network storage. In *Proceedings of Workshop on Internet Service Quality Economics*, Cambridge MA, Dec. 1999.
- [10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. MIT Press, Cambridge, Massachusetts, 2001.
- [11] J. Edler and M. D. Hill. Dinero IV Trace-Driven Uniprocessor Cache Simulator, 1998.
- [12] K. Fall. Network emulation in the Vint/NS simulator. In *Proceedings of IEEE Symposium on Computers and Communications (ISCC'99)*, July 1999.
- [13] L. Fan, P. Cao, J. Almeida, and A. Z. Broder. Summary Cache: A scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.
- [14] D. P. Helmbold, D. D. E. Long, and B. Sherrod. A dynamic disk spin-down technique for mobile computing. In *Proceedings of the 2nd Annual International Conference on Mobile Computing and Networking 1996 (MOBICOM '96)*, pages 130–142, Rye, New York, Nov. 1996. ACM.
- [15] M. Herbster and M. K. Warmuth. Tracking the best expert. In *Proceedings of the 12th International Conference on Machine Learning*, pages 286–294, Tahoe City, CA, 1995. Morgan Kaufmann.
- [16] M. D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. PhD thesis, University of California, Berkeley, November 1987.
- [17] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. Wes. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):51–81, Feb. 1988.
- [18] S. Jin and A. Bestavros. GreedyDual* web caching algorithm: Exploiting the two sources of temporal locality in web request streams. In *Proceedings of the 5th International Web Caching and Content Delivery Workshop*, Lisbon, Portugal, May 2000.
- [19] T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th Conference on Very Large Databases (VLDB)*, pages 439–450, Santiago, Chile, 1994.
- [20] J. M. Kim, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *Proceedings of the 4th Symposium on Operating Systems Design and Implementation (OSDI)*, pages 119–134, San Diego, CA, Oct. 2000.
- [21] P. Krishnan, D. Raz, and Y. Shavitt. The cache location problem. *IEEE/ACM Transactions on Networking*, 8(5):568–582, 2000.
- [22] J. Kubiawicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. OceanStore: An architecture for global-scale persistent storage. In *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Cambridge, MA, Nov. 2000. ACM.
- [23] N. Littlestone and M. K. Warmuth. The Weighted Majority Algorithm. *Information and Computation*, 108(2):212–261, Feb. 1994.

- [24] N. Megiddo and D. S. Modha. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *Proceedings of the 2003 Conference on File and Storage Technologies (FAST)*, pages 115–130, San Francisco, CA, Mar. 2003.
- [25] E. J. O’Neil, P. E. O’Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 297–306, 1993.
- [26] L. Rizzo and L. Vicisano. Replacement policies for a proxy cache. *IEEE/ACM Transactions on Networking*, 8(2):158–170, 2000.
- [27] A. Silberschatz, P. Galvin, and G. Gagne. *Applied Operating System Concepts*. John Wiley & Sons, Inc., first edition, 2000.
- [28] X. Tang and S. T. Chanson. Coordinated en-route web caching. *IEEE Transactions on Computers*, 51(6):595–607, June 2002.
- [29] V. Vovk. Aggregating strategies. In *Proceedings of the 3rd Annual Workshop on Computational Learning Theory*, pages 371–383, Rochester, NY, 1990. Morgan Kaufmann.
- [30] D. A. B. Weikle. *Caches As Filters: A Framework for the Analysis of Caching Systems*. PhD thesis, University of Virginia, 2001.
- [31] G. Weikum, A. C. Konig, A. Kraiss, and M. Sinnwel. Towards self-tuning memory management for data server. *Data Engineering Bulletin*, 22(2):3–11, 1999.
- [32] C. Williamson. On filter effects in web caching hierarchies. *ACM Transactions on Internet Technology (TOIT)*, 2(1):47–77, 2002.
- [33] D. L. Willick, D. L. Eager, and R. B. Bunt. Disk cache replacement policies for network file servers. In *Proceedings of the 13th International Conference on Distributed Computing Systems (ICDCS ’93)*, pages 2–11, Pittsburgh, Pennsylvania, May 1993.
- [34] T. M. Wong and J. Wilkes. My cache or yours? making storage more exclusive. In *Proceedings of the 2002 USENIX Annual Technical Conference*, pages 161–175, Monterey, CA, June 2002. USENIX.
- [35] R. Wooster and M. Abram. Proxy caching that estimates page load delays. In *Proceedings of the 6th International World Wide Web Conference*, pages 325–334, Santa Clara, CA, Apr. 1997.
- [36] Y. Zhou, J. F. Philbin, and K. Li. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the 2001 USENIX Annual Technical Conference*, pages 91–104, Boston, Massachusetts, June 2001. Usenix.