视频点播环境下的缓存算法研究

钱培杰[1] 武 娟[2] 高成英[1]

(中山大学软件学院 广州 510006)[1] (中国电信股份有限公司广州研究院 广州 510630)[2]

摘 要 在视频点播环境下,用户的视频访问速度的提高是提升用户体验的关键。用户的视频访问速度与服务器响应速度、网络传输等相关,其中服务器对用户请求的响应是主要因素。作为视频点播环境下提高用户访问速度的一种重要手段,缓存技术一直是工业界和学术界关注的焦点。针对LRU、LRU、LRFU、SC等经典算法进行对比分析,并结合在视频点播环境下的模拟数据和某运营商提供的实际数据进行实证研究,观察各算法的实际表现。分析了各算法的应用结果,发掘视频点播环境下缓存算法的选择策略,为提升视频点播系统的缓存命中率提供了理论依据。

关键词 视频点播系统,缓存算法,模拟实验,实证研究

中图法分类号 TP311 **文献标识码** A

Study and Implementation on Cache Algorithms

QIAN Pei-jie[1] WU Juan[2] GAO Cheng-ying[1]

(School of Software, Sun Yat-sen University, Guangzhou 510006, China)[1]

(Guangzhou Research Institute of China Telecom Corporation Limited, Guangzhou 510630, China)[2]

Abstract In video on demand system, the speed of users accessing the videos is the key to improve the user experience. The speed of the users accessing the videos is related to the response speed of the servers, the net transmission and so on, among which the response speed of the servers is the main factor. As an important application to improve the user access speed in video on demand system, the cache technology attracts much attention in industry and academia. This paper focused on the comparison and analysis of LRU, LFU, LRFU, SC and some other classic algorithms. Simulated data and actual data were both used to demonstrate and research. The actual performance of the algorithms was observed in order to help select the right cache algorithm in video on demand system, which provides theory evidence to improve the hit ratio of cache in video on demand system.

Keywords Video on demand system, Cache algorithms, Simulation experiment, Empirical research

1 引言

随着网络与多媒体技术的发展,视频点播在各个平台上的应用也越来越广泛,比如IPTV、视频网站等,人们对视频点播的要求也越来越高,其中,视频点播速度的提高是一大关键。视频点播环境下的缓存技术是提升用户访问速度的重要环节。视频提供商为了提高视频点播质量,会设立缓存服务器来存储一些热门的资源,如基于CDN的流媒体系统,如图1所示。

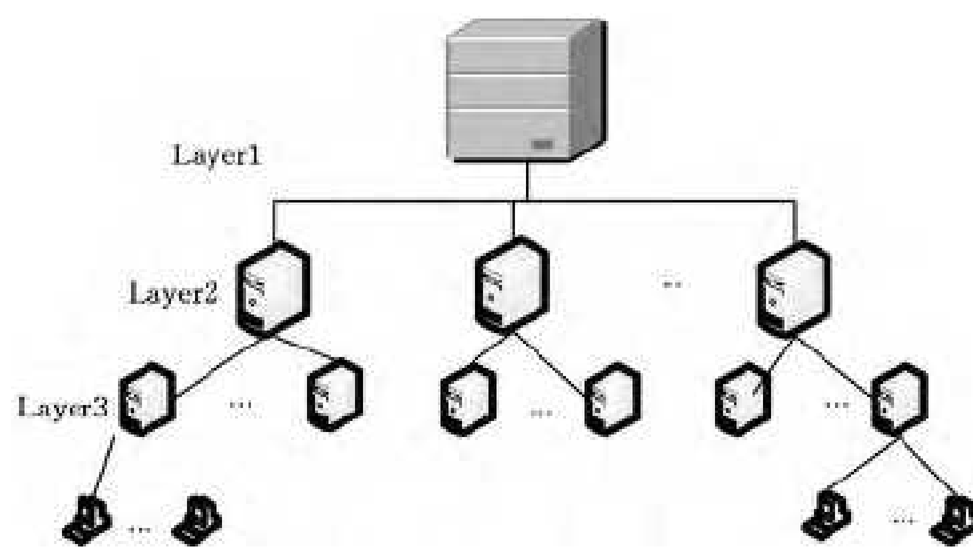


图1 基于CDN的流媒体系统

基于CDN的流媒体系统将服务器分为3层,每一层都存储不同热度的资源,用户请求将被发至最近的缓存服务器,如果请求的资源在这台服务器上,服务器即可发回资源给用户,否则服务器将请求向上发送,直至找到用户请求的资源。因此,为了减轻根服务器的负载并节省网络传输成本,缓存服务器应当具有预估用户请求的能力,即能够根据用户的请求历史判断最近的热门资源,并使用一定的缓存替换算法更新缓存服务器上的资源,从而快速响应用户请求,提升用户体验。

缓存算法发展至今,已有一些令人较为满意的成果,从一开始的包括基于访问频率的算法,到基于访问时间的算法,再慢慢完善,现在已有了访问时间与访问频率相结合的算法等较为理想的算法。基于访问频率的算法,通过在某段时间内对资源被访问的次数进行统计,来判断该资源接下来会不会被访问。这类算法理所当然,最容易理解,因此也最容易被接受,包括LFU(Least Frequently Used),以及在此基础上发展起来的2Q[1](2 Queues)、LIRS[2](Low Inter-Reference Recency Set)等;基于访问时间的算法,记录资源访问的时间,以时间作判断依据,包括LRU(Least Recently Used);访问时间

本文受国家自然科学基金(61472455),广东省自然科学基金(2014A030313154)资助。

钱培杰 硕士生,主要研究方向为网络多媒体技术;武 娟 女,硕士,高级工程师,主要研究方向为云计算、互联网及电信网络技术、互动新媒体技术等;高成英 女,博士,副教授,主要研究方向为计算机网络、图形图像处理与传输。

与访问频率相结合的算法，综合了基于访问频率和访问时间的算法，包括 LRFU[3]（Least Recently Frequently Used）等。当然，因为现实当中有各种各样的因素，这些缓存算法各自或多或少地存在一些优点和缺点，因此，没有一种缓存算法能完美解决所有的缓存问题。结合实际情况，通过对已知的缓存算法进行改善，使之能切合我们的目标，才是关键所在。在现实因素的干扰下，以上提及的缓存算法各自或多或少地存在一些优点和缺点，因而拥有自己的使用范围，并且需要根据具体情况进行调整。比如，LRU 实现简单但依赖服务器存储容量，LFU 依赖数据本身的规律，LRFU 适应能力差等。文献[6]重点研究了视频点播系统下的 LFRU 并作出改进，使得缓存替换效率得到了提高。本文针对 VOD 系统，根据用户访问模式与其他系统不同的特点，探索不同缓存算法的效果，并给出缓存替换策略。

2 算法原理与实现

缓存算法的发展已有一定的历史，许多算法是在以往的基础上进行改进而形成的。经典算法如 LRU 和 LFU，实现简单，是最基础的算法。LRU 改进算法如 LRU-K[4]、2Q 和一些变种算法从不同方面对 LRU 进行了不同的改进，均有一定的成效。将 LRU 和 LFU 结合起来的 LRFU，兼顾 LRU 和 LFU，同时具备 LFU 和 LFU 的优势。其他新颖算法如 SC[5]（Scoring based Caching）和 LIRS 运行稳定，在特定的环境下表现优异。

2.1 最近最少使用算法（LRU）

LRU，顾名思义，缓存中将保留最近一段时间内经常使用的数据，而淘汰最近未被经常使用的数据。LRU 基于这个一个事实：在最近一段时间内经常被使用的数据在未来一段时间内也会被使用，而未被经常使用数据在未来很长时间内不会被用到。因此，在替换内容时只需要找出最近最少使用的那些数据进行替换即可。

LRU 在缓存中维持一个内容列表，用于存储最近被使用的数据。当有数据被请求时，它将跃至第一位。如果被请求的数据在列表中，即已经在缓存中存储（比如排在第 k 位），那么排在 k 之前的数据位置将拉到向后一位，没有内容会被淘汰；如果被请求的数据不在列表中，它将被提取到缓存中，跃至第一位的同时，其他内容的位置顺次拉后一位，排在最后一位的内容将被淘汰。

如图 2 所示，假设有 1—10 10 个内容，底层服务器只够容纳 5 个内容，初始时为 1，2，3，4，5 这 5 个内容。第 1 个用户请求内容 2，内容 2 在服务器中存在，内容次序改为 2，1，3，4，5；第 2 个用户请求内容 6，内容 6 在缓存中不存在，缓存向高级服务器拉取内容 6，并将内容 6 提至第一位，次序变为 6，2，1，3，4，原来的内容 5 将被淘汰。

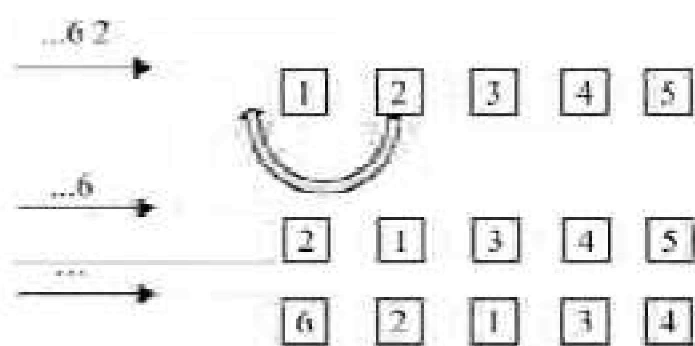


图 2 LRU 示例图

LRU 算法遵循这样的一个原则：最近被访问的内容在未来访问的几率增大，于是才出现了经典的 LRU 算法。LRU 算法的提出是显然的，它基于内容访问热度来对内容进行取舍，留下访问次数多的，剔除访问次数少的，这符合多数人的设想。然而，不难看出，该算法具体的操作过程仍然需要研究和改进。经典的 LRU 算法单纯地从最近的访问次数出发，直接将最近一次被访问的内容的优先级提至最高。然而，最近被访问的内容不一定是最热门的，这将可能导致对内容热门程度判断失误的情况，从而影响用户访问内容和服务器响应用户的效率。因此，从这种角度上来看，这种算法仍有待改善。

具体实现：服务器存储内容使用一个链表数组 *contents—server* 表示。每当内容 X 收到访问请求时，检查 X 是否在 *contents—server* 内，如果有，将 X 放到 *contents—server* 的首位；否则，创建 X 放到 *contents—server* 的首位，并移除 *contents—server* 的末尾。伪代码如下：

```
IF X is in contents—server
  put X on the top of contents—server
ELSE
  put X on the top of contents—server
  remove the end of contents—server
END ELSE
END IF
```

2.2 最少频率使用算法（LFU）

在 LRU 算法中，最近访问的时间是考虑的重要因素。而 LFU 算法从频率上出发，按照内容访问频率对内容进行排序，这比单纯地考虑访问时间要合理得多，毕竟访问频率才是能够反映内容热度的一个重要的标准。在随后的很多算法中也采取了与访问频率相关的方法。另外，LFU 算法也使用了不同于 LRU 算法的缓存机制。

LFU 在缓冲中维持一个总数列表，记录视频的被访问次数。当请求的视频未在缓存中时，根据访问总数列表淘汰掉缓存中访问次数最少的视频。

如图 3 所示，初始时缓存中没有视频，访问次数列表中的内容的访问次数均为 0。每当有一个内容的请求到来，该列表就更新一次，在这个内容的访问次数上加 1。当访问的内容未命中时，将缓存中访问次数最少的视频替换出去。

从理论上来看，LFU 算法从内容的访问频率上对内容进行排序，留下访问频率高的，替换掉访问频率低的。因为访问频率与内容热度有很大的关联，因此，热度高的内容将会有很大的几率保存下来，这正是缓存算法优化的目的。

LFU 算法从另一个角度出发，考虑了内容被访问的次数，即内容被访问的频率，显然这是符合常理的，能够反映内容的热门程度。然而，LFU 算法也有其难点所在，比如计算频率的时间段 T 的控制，替换的时间设定等。总之，在不同场合，该算法都需要进行相应的改进。

具体实现：服务器存储内容使用一个链表数组 *contents—server* 表示，访问总数列表使用一个链表数组 *freq* 表示。每当内容 X 收到访问请求时，更新 *freq*，在 X 对应的访问次数上加 1。检查 X 是否在 *contents—server* 内，如果没有，则移除 *contents—server* 中访问次数最少的视频，并将 X 加入到 *contents—server* 中。伪代码如下：

```
freq[X] → freq[X]+1
IF X is not in contents—server
  remove the video i with the least frequency
END IF
```

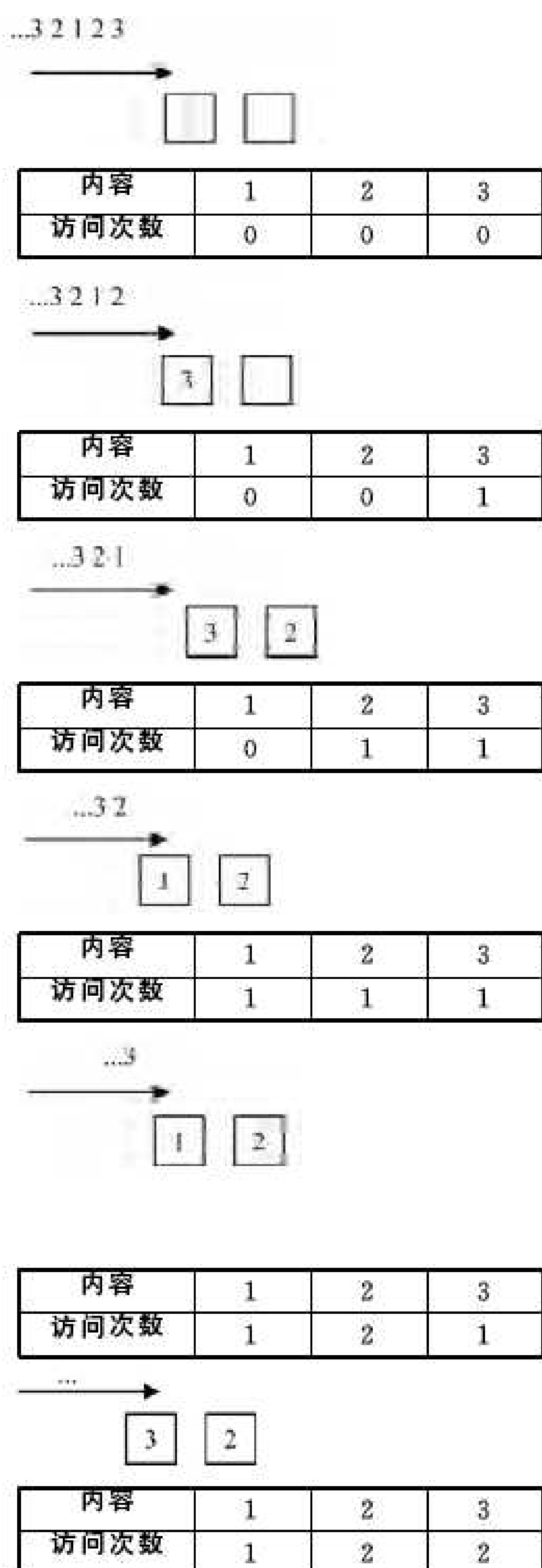


图 3　LFU 示例图

2.3　最近最少使用-K 算法(LRU-K)

在 LRU 中,只要资源被访问一次,那么它将在缓存队列中直接排至首位。从某种意义上看,这样做并不合理,因为被访问的资源在未来一段时间内有可能不被访问到,这样做将导致缓存资源被占用,致使其他内容无法进入缓存。一种解决的办法是:在限定的 T 时段内,当某资源被访问 K 次后,才将它放入缓存队列的首位。于是,有了 LRU-K 算法。

如图 4 所示,设置一张频率列表记录每个内容在时段 T 内的访问频率。T 时段内,当访问的内容不小于 K(此处为 2)次时才将该内容放到首位。此处,内容 2 的访问频率为 2,符合要求,因此将它放到首位。

LRU-K 是对 LRU 作出的调整,它弥补了 LRU 对待新访问的内容过于宽松的缺点,限制了部分内容对缓存的占有,给其他内容提供了空间。然而,参数 K 是关键,需要结合实际经过大量实验进行确定。

具体实现:服务器存储内容使用一个链表数组 *contents—server* 表示,内容访问频率使用一个链表数组 *contents—access* 表示。每当内容 X 收到访问请求时,X 在 *contents—access* 中对应的频率将增加,同时检查 X 是否在 *contents—server* 内,如果有,将 X 放到 *contents—server* 的首位;否则,检查 X 的频率是否大于 K,如果大于 K 则将 X 替换进 *contents—server*,移除 *contents—server* 的末尾。伪代码如下:

```
IF X is in contents—server
contents—access[X] → contents—access[X]+1
  IF contents—access[X] ≥ K
    remove X from contents—server
    insert X on the top of contents—server
  END IF
  IF tk ≥ Tk
    clear contents—access
tk=0
ELSE
contents—access[X] → contents—access[X]+1
  IF contents—access[X] ≥ K
    remove the end of contents—server
    insert X on the top of contents—server
  END IF
  IF tk ≥ Tk
    clear contents—access
    tk=0
  END IF
END ELSE
END IF
```

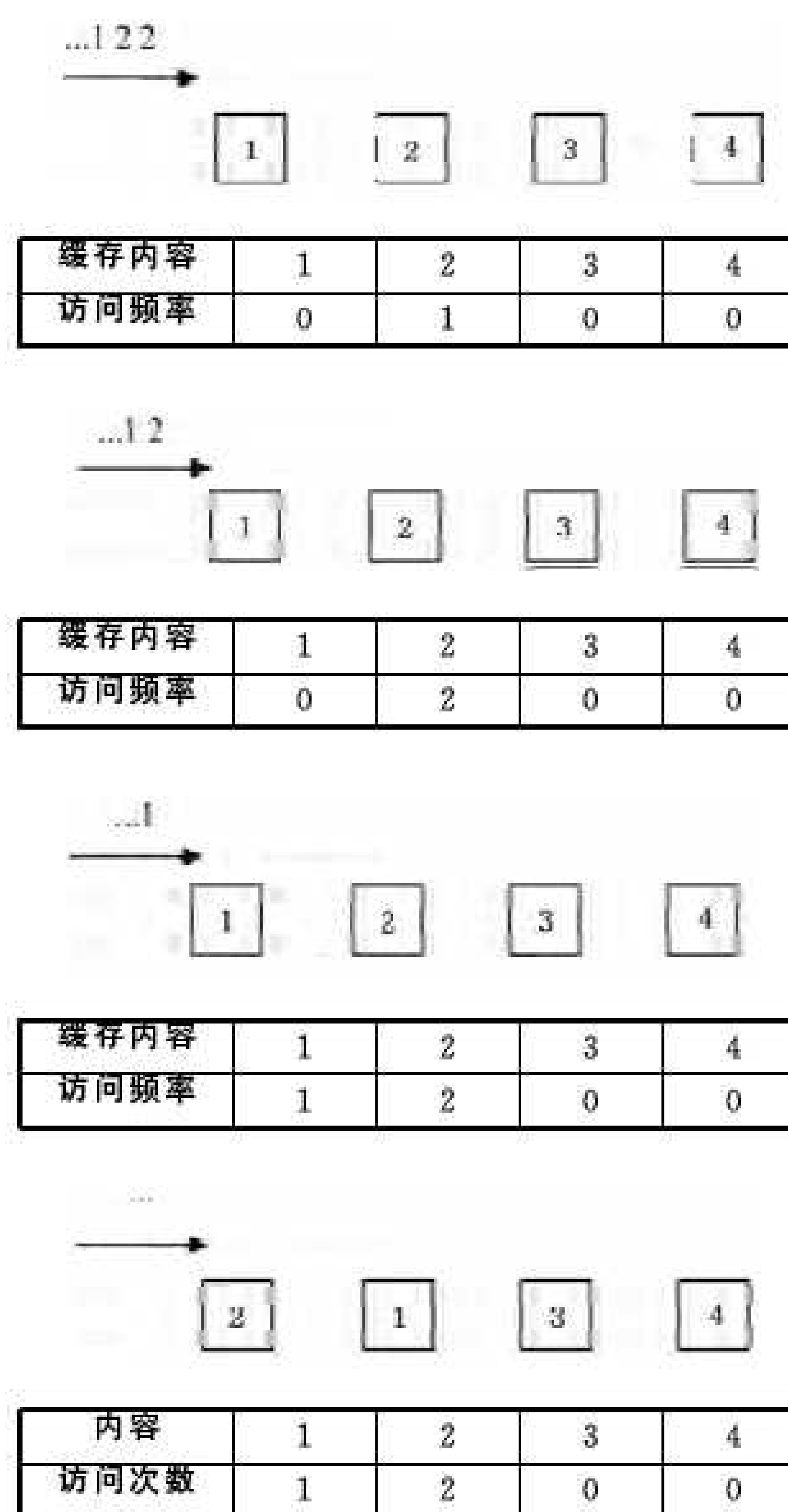


图 4　LRU-K 示例图

2.4　最近最少使用变种算法(LRU—VAR)

LRU 的一个最大缺陷是将最近访问的内容的次序直接提至第 1 位,这是不合理的,因为最近被使用的内容却不一定是最受欢迎的,提至第 1 位的做法略为不妥。于是,从这个角度出发,可以对该算法进行一些改进,因此产生了 LRU 的变种——LRU—VAR。

在 LRU 算法中,每当用户访问一个内容时,该内容的次序提至第 1 位,而在该算法的变种中,被访问的内容的次序将不再被提至第 1 位,而被提至第 J 位,排在第 J 位之后的内容次序依次增加,末位被替换。

如图 5 所示,初始缓存队列为 1,2,3,4,当请求内容 4 时,内容 4 上升 J($J=2$)位,当请求内容 3 时,内容 3 上升 J($J=$

2)位，而不再是原来 LRU 的直接排至首位的做法。

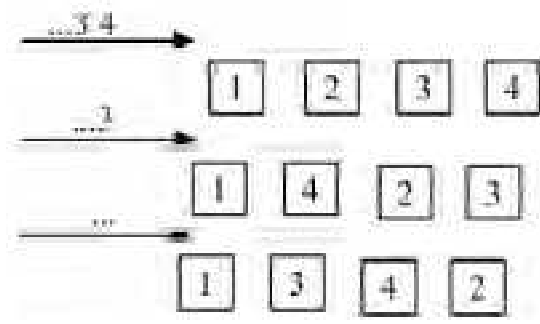


图5　LRU—VAR 示例图

LRU—VAR 是对 LRU 的另一种改进方法，这种改进可在某种程度上提升 LRU 算法的效率，毕竟这使得"最近最少使用"算法不再那么紧张，在内容排序上有了一定舒缓的空间，最近依次被访问的内容有了更多的上升空间，其他未被访问的内容也有更多变化的可能。

2.5　基于分数的算法(SC)

SC 是基于分数计算的方法。在 SC 中，每个内容都会有属于自己的一个"分数"。初始时，每个内容都有一个初始"分数"，当某个内容被访问时，它的"分数"将增加一个数值，而其他未被访问的"分数"将减少一个数值。当需要进行缓存替换时，"分数"最低的内容将被替换掉。

如图6所示，设置一张分数表记录每个内容的"分数"。每个内容的初始"分数"为5，当内容3被访问时，它的"分数"将增加1，其他内容的"分数"将减少1；同样，当内容4和5到来时，它们的"分数"也将增加1，其他内容的"分数"也将减少1。最后，当内容2被请求时，它的"分数"增加1变成3，其他内容的"分数"减少1，"分数"最低的内容1被淘汰。

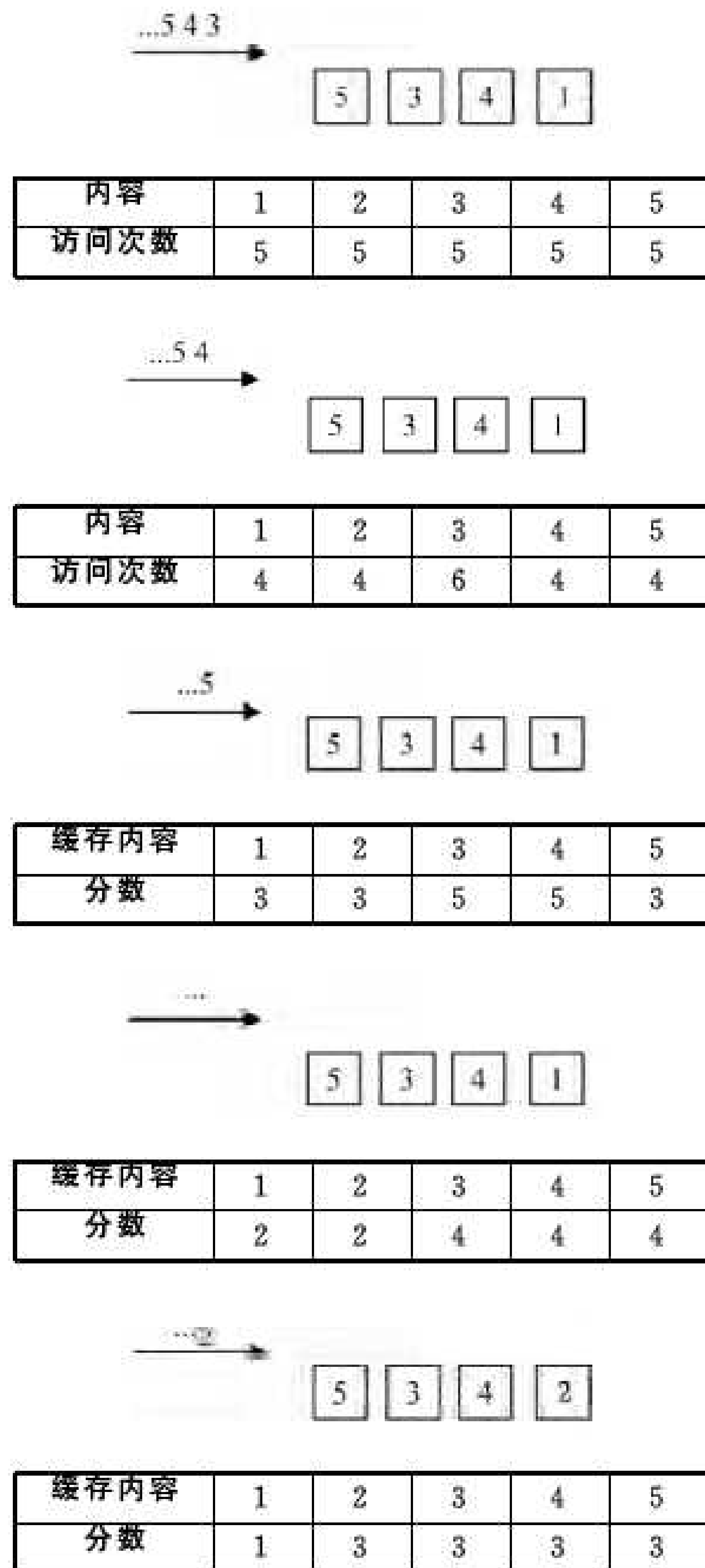


图6　SC 示例图

SC 本质上是对 LRU 的一种改进，在考虑最近一次访问的内容被命中概率的同时削弱其他内容被命中的概率。然而，在 SC 的实现过程中，对于初始分数、增加的分数、减少的分数的界定是 SC 成败的关键，需要通过大量数据不断试验调整才能得到较好的参数。

具体实现：服务器存储内容使用一个链表数组 *content—server* 表示，每个内容的"分数"由一个数组 *scores* 表示，初试分数由 *initSc* 表示，增加的分数由 *inSc* 表示，减少的分数由 *deSc* 表示。每当内容 X 收到访问请求时，根据 SC 算法原理对所有内容进行对应的"分数"更新。伪代码如下：

```
IF X is in contents—server
  scores [X] → scores [X]+inSc
  for i=0 to the end of contents—server
    IF contents—server[i] != X
      scores[contents—server[i]]→ scores[contents—server[i]] —
      deSc
    END IF
ELSE
  scores[X]=initSc
  for i=0 to the end of contents—server
    scores[contents—server[i]]—= deSc
  sort the contents according to the scores
  IF the score of the last one in server < initSc
    remove the last one of contents—server
    insert X on the top of contents—server
  END IF
END ELSE
END IF
```

2.6　双队列算法(2Q)

2Q 算法，顾名思义，使用两个队列记录数据的访问历史。假设使用的两个队列分别为队列1和队列2。每当有内容请求访问时，首先检查队列1有没有该内容，如果有，则将该内容放到首位；如果没有，检查队列2有没有该内容，如果队列2有该内容，该内容从队列2移到队列1首位，并将队列1的末位内容放到队列2的首位；否则，将它加入到队列2的首位，并移除掉队列2的末位。

如图7所示，当服务器收到内容2的请求时，检查两个队列，发现队列1存在内容2，于是将内容2放到首位；当收到内容6的请求时，检查两个队列，发现队列1不存在内容6，继续检查队列2，发现队列2存在内容6，于是将队列2中的内容6放到队列1的首位，将队列1末位的内容4放到队列2的首位；当收到内容9的请求时，检查两个队列发现均不存在内容9，于是将内容9放到队列2的首位，队列2的末位内容8被淘汰。

图7　2Q 示例图

2Q 算法本质上也是对 LRU 的改进，通过双队列的方式确定某个内容最近被访问的频繁程度，从而间接反映其热门

程度。双队列的设置减少了缓存内容缓存的时间且提高了中率相关性，从而提高了缓存命中率。在 2Q 的基础上可进行拓展，成为 $kQ(k\geqslant 2)$，即设置多个队列，其原理和过程与 2Q 类似，然而，队列数量的多少也需要通过大量实验来确定。

具体实现：服务器存储内容使用一个链表数组 *contents—server* 表示，另一个缓存队列用 *histque* 表示。每当有内容 X 的访问请求时，服务器检查两个队列，并根据 2Q 的原理对两个队列中的内容进行更新。伪代码如下：

```
IF X is in contents—server
  remove X from contents—server
  insert X on the top of contents—server
ELSE
  IF X is in histque
    remove X from histque
    insert the end of contents—server to to histque
    remove the end of contents—server
    insert X on the top of contents—server
  ELSE
    insert X on the top of histque
    remove the end of histque
  END ELSE
  END IF
END ELSE
END IF
```

2.7 最近最少使用和最少频率使用结合算法(LRFU)

LRFU 结合了 LRU 和 LFU，通过调整这两个算法各自的比例，融合成 LRFU。对于每个内容 x，初始时都有 $C(x)=0$。当有内容被请求访问时，更新每个 $C(x)$。如果 x 是被请求访问的内容，则 $C(x)=1+2^{\wedge}(-\lambda)C(x)$，否则 $C(x)=2^{\wedge}(-\lambda)C(x)$（如下所示）。每次要替换内容时，$C(x)$ 值最小的内容将被替换掉。

$$C(x)=\begin{cases}1+2^{\wedge}(-\lambda)C(x), & x \text{ is referenced} \\ 2^{\wedge}(-\lambda)C(x), & \text{else}\end{cases} \quad (1)$$

由式(1)可知，当 λ 趋于 0 时，$C(x)$ 趋于 x 的次数，LRFU 趋于 LFU；当 λ 趋于 1 时，$C(x)$ 趋于 x 最近一次访问的时间，LRFU 趋于 LRU。因此，λ 将在 [0,1] 中适当调整，λ 的调整影响着 LRFU 的效率。

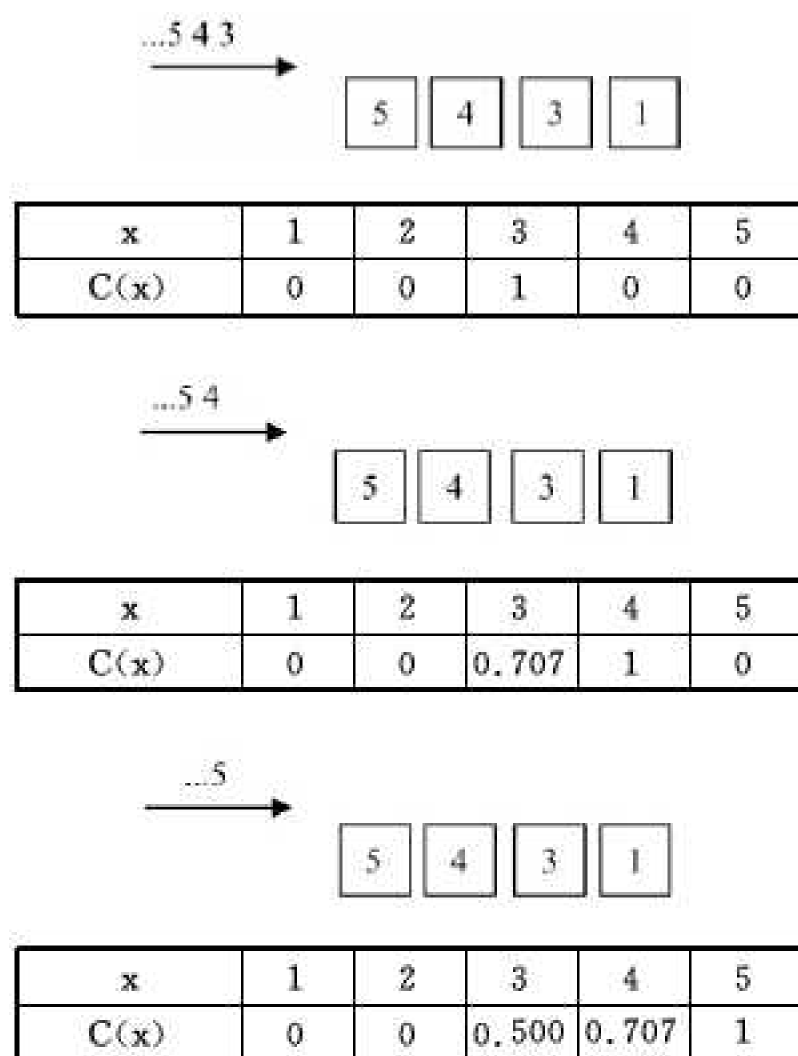


图 8 LRFU 示例图

如图 8 所示($\lambda=0.5$)，根据式(1)，每次有内容被请求访问时，更新所有内容的 $C(x)$，最后当内容 5 被请求访问时，根据 $C(x)$ 的大小按照次序淘汰内容 2，用内容 5 进行替换。

LRFU 综合了 LRU 和 LFU 的特点，由参数 λ 决定两种算法所占比例，利用 LRU 和 LFU 的优点提高缓存命中率。参数 λ 的确定决定了该算法是否能有效地运行，另外，由于每次都需要对每个内容的 $C(x)$ 进行更新，因此该算法的效率比其他算法低。

具体实现：服务器存储内容使用一个链表数组 *contents—server* 表示，每个内容的“分数”由数组链表 *cx* 表示。每当有内容 X 的访问请求时，根据 LRFU 的分数更新规则，对每个内容的“分数”进行更新。当需要进行缓存替换时，分数最低的内容将被替换出去。伪代码如下：

```
IF X is in contents—server
  for i=0 to the end of contents—server
    IF contents—server[i] == X
      cx[X] → 1+cx[X]*2^(−λ)
  ELSE
    cx[i]→ cx[i]*2^(−λ)
  END ELSE
  END IF
ELSE
  for i=0 to the end of contents—server
    cx[i] → cx[i]*2^(−λ)
  cx[X] → 1+cx[X]*2^(−λ)
END ELSE
END IF
key → the least one in cx
remove the key from contents—server
insert X to contents—server
```

2.8 最短最近使用间隔算法(LIRS)

LIRS 是对 LRU 的改进算法。在 LIRS 中，有两个概念极其重要：

- IRR(Inter-reference Recency)，指同一个内容两次被访问时间之间其他不同内容的数量。
- Recency，指不同内容与当前时间之间内容的数量。

LIRS 是根据 IRR 和 Recency 进行缓存选择的。根据每个内容的 IRR，LIRS 将所有内容分成 HIR(High Inter-reference Recency)和 LIR(Low Inter-reference Recency)。HIR 中又分为存在缓存中的和不存在缓存中的。LIRS 尽力将 LIR 中的内容留在缓存中，因此，它被称为 LIRS。

											Recency	IRR
E									x		0	inf
D		x					x				2	3
C				x							4	inf
B			x		τ						3	1
A	x					x		x			1	1
Blocks / Virtual time	1	2	3	4	5	6	7	8	9	10		

图 9 LIRS 示意图

如图 9 所示，内容 A 的最后两次访问中间存在一次对内容 D 的访问，因此内容 A 的 IRR 是 1，A 最后一次访问距离当前时间中间只有 1 个内容，因此，内容 A 的 Recency 为 1。根据 IRR 的大小，最后将所有内容分成 $LIR=\{A,B\}$，$HIR=\{C,D,E\}$，由于 HIR 的容量为 1，因此，在 HIR 中根据 Recency 进行替换，最终留下了 E。因此，最后内容的分布为 $LIR=\{A,B\}$，$HIR=\{E\}$。

LIRS 的实现细节：设置两条队列即 S 和 Q，其中 S 存储

每次访问的内容，Q存储缓存中的HIR。将下列行为定义为“栈修剪”：移除S底部的内容，保证S的底部一直都是LIR。所有的内容将被分为LIR或者HIR。而HIR又被分为缓存中的HIR和不在缓存中的HIR。队列S存储LIR、缓存中的HIR和不在缓存中的HIR，队列Q存储缓存中的HIR。对于每次内容访问，根据下列的情况分别处理：

1. 访问到LIR内容X：这种情况属于命中缓存，将内容X移到队列S的首位，如果内容X是在S的底部，进行“栈修剪”。

2. 访问到缓存中的HIR内容X：这种情况属于命中缓存，将内容X移到队列S的首位。这时会有两种情况：1）如果X是在S中，将它的状态改为LIR，并从Q中移除出去，S底部的LIR内容状态改为HIR，移到Q的末尾；2）如果X不在S中，将它的状态改为HIR，并移到Q的末尾。

3. 访问到不在缓存中的HIR内容X：这种情况属于非命中缓存，移除Q的头部，将它替换出缓存，并将X替换进缓存，放到S的头部。这时会有两种情况：1）如果X在S中，将它的状态改为LIR，并将S末尾的LIR内容状态改为HIR，放到Q的末尾，然后进行“栈修剪”；2）如果X不在S中，将它的状态改为HIR，并放到Q的末尾。

具体实现：使用数组链表S和Q分别表示LIRS原理中使用得到的两个链表。每当有内容X的访问请求时，判断X在S和Q中的状态，根据LIRS原理对S和Q的内容进行更新。伪代码如下：

```
conduct stack pruning: make sure the last one of S is LIR
IF X is in S
  IF X in S is LIR
    remove X to the top of S
    conduct stack pruning
  ELSE IF X in S is in the cache
    remove X to the top of S
    IF X is in Q
      remove X from Q
    remove the end of S to the top of Q
    conduct stack pruning
  ELSE
    remove X to the top of S
    remove the end of Q
    remove the end of S to the top of Q
    conduct stack pruning
  END ELSE
  END IF
ELSE
  IF X is in Q
    remove X to the top S and Q
  ELSE
    remove the end of Q
    insert X to the top of S
    insert X to the top of Q
  END ELSE
  END IF
END ELSE
END IF
```

2.9 各算法比较

上述各缓存算法具有各自的优劣之处，如表1所列。

表1 算法比较

缓存算法	优点	缺点
LRU	实现简单，效率较高	依赖服务器存储容量
LFU	实现简单，效率稳定	依赖服务器存储容量；依赖数据本身的规律
LRU-K	LRU改进算法，效率较LRU高	运行速度慢；摆脱不了LRU固有的缺点
LRU-VAR	LRU改进算法，效率较LRU高	同上
2Q	LRU改进算法，效率在类LRU算法中最高	同上
LRFU	在数据有明显规律地分布时表现较好	参数不容易权衡，运行效率较低
SC	自适应能力强	同上
LIRS	同上；在所有算法中表现最稳定	实现复杂，难以调试

3 仿真/实验结果与分析

3.1 模拟数据仿真结果与分析

相关研究表明，基于VOD系统的用户访问模式表现出一定的规律性。以IPTV为例，通过收集在IPTV中用户的真实记录，对用户的行为特征进行统计分析。用户的行为特征包括访问模式、视频热度、交互模式、播放时长等。通过分析用户行为，可以优化视频调度、存储策略。这里主要涉及对视频热度和播放时长的分析。

热度分析：对于点播视频，Zipf分布比几何分布更适合对视频热度建模，如图10所示。

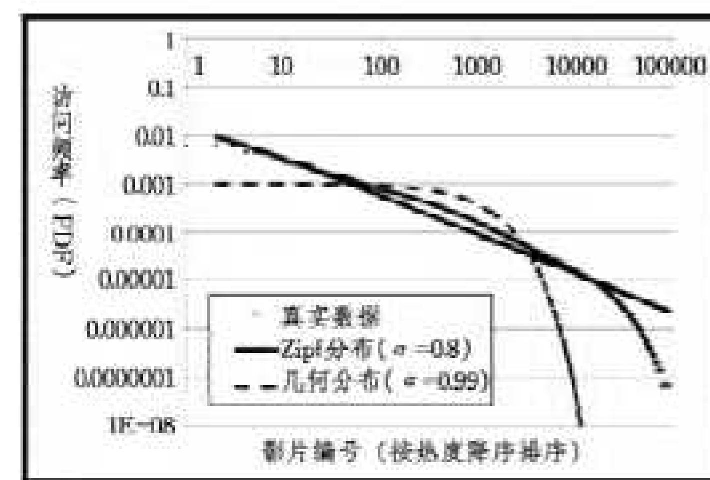
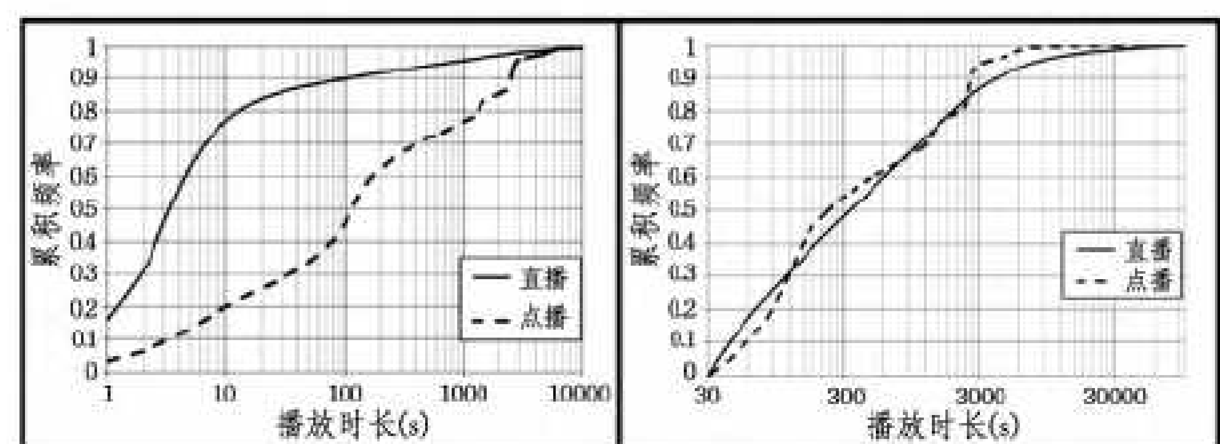


图10 热度匹配Zipf分布与几何分布

用户播放时长分析：播放时长表示为观看一个直播频道或点播视频的连续时间（直到切换到另外一个频道或者视频），如图11所示，约80%用户直播和点播播放时长少于总时长的10%。



(a)全部记录　　(b)大于30秒记录

图11 播放时长累积频率分布图

根据以上对视频热度和播放时长的研究，模拟数据将具有以下特点：

①视频的访问频率遵从Zipf分布规律。

$$P(\gamma)=\frac{C}{\gamma^{\alpha}} \tag{2}$$

式中，γ表示一个视频出现频率的排名，$P(\gamma)$表示排名为γ的视频出现的频率。

②每个视频被分成10段，在模拟请求中，视频的第1段的访问次数将占整个视频访问次数的80%。

使用模拟数据检验各缓存算法，得到各算法在不同的服务器容量下命中率比较，如表2所列。

表2 模拟数据下缓存算法命中率比较列表

缓存容量占总量比例	LRU	LFU	LRFU	LRU—VAR	SC	2Q	LRU-K	LIRS
0.005	0.598	0.718	0.720	0.600	0.720	0.640	0.600	0.688
0.0075	0.674	0.776	0.777	0.674	0.777	0.708	0.0674	0.720
0.00875	0.703	0.800	0.800	0.703	0.800	0.733	0.0703	0.699
0.01	0.730	0.813	0.814	0.7301	0.814	0.758	0.730	0.740
0.012	0.989	0.804	0.989	0.989	0.982	0.930	0.981	0.944

由表2得到以下结论：

1）随着缓存容量的增加，各类缓存算法的命中率有所增加。这反映出服务器缓存容量对缓存算法的影响，是系统性能的关键。

2）LRU及其改进算法LRU—K和LRU—VAR算法表现一致，也同时是最差的，可见单纯的LRU算法并不适用于视频点播系统；2Q和LIRS算法表现较好，它们的命中率十分接近，而且在过程中，它们的运行速度较快，在视频点播环境下是性价比较高的缓存算法。

3）LFU、LRFU和SC算法表现最好，它们的命中率很高，而且十分接近，但是由于LRFU和SC算法需要对所有的内容的相关记录进行更新，运行速度较慢，效率没有LFU高。

4）LFU是命中率最高的3个算法之一，而且由于其易实现、效率高的特点，如果需要考虑运行内存、运行效率等问题，LFU无疑是最好的选择。

5）所有算法都有自适应调整的能力，服务器存储的内容初始是随机的，随着请求数量的增加，缓存最后都能调整好并有很高的命中率。

3.2 实验数据仿真结果与分析

除了使用模拟的数据，还代入了电信提供的真实数据。下面为使用的有关真实数据的具体情况（该数据来源于2012年6月10日中国电信运营商CDN系统）。

数据情况：

内容总量：47151

请求总量：1696090

缓存容量占总量比例：0.01

使用实际数据的实验结果如表3所列。

表3 实际数据下缓存算法命中率比较

缓存容量占总量比例	LRU	LFU	LRFU	LRU—VAR	SC	2Q	LRU-K	LIRS
0.01	0.719	0.738	0.738	0.719	0.738	0.743	0.720	0.641

观察各算法运行情况，可得到以下结论：

1）LRU、LRU—VAR和LRU—K在VOD系统上的表现差别不大，可见在VOD系统上LRU并不是最好的选择。

2）LFU的表现比LRU要好。

3）LRFU、LFU的表现相似，都是命中率比较高的算法，可见在VOD系统上LFU算法的重要性比LRU大。

4）SC和LRFU、LFU的表现一样，是命中率较高的算法。

5）2Q使用双队列的方法提高了命中率，它是对LRU的最好改进，其表现是所有算法中最好的。

6）LIRS的表现较差，命中率是最低的，不适用于VOD系统。

3.3 总结分析

通过对模拟数据和实际数据的实验分析发现，模拟数据和实际数据的实验结论基本吻合，除了在LIRS算法上有所出入外，其他算法的比较情况基本一致，这说明模拟的数据和实际数据十分切合，是对实际情况的真实反映。在视频点播环境下，LFU、LRFU和SC是最好的3种算法，如果仅考虑缓存命中率，LRFU是最好的选择；如果综合考虑服务器容量、网络成本等问题，则可以具体深入研究这3种算法之间的差异，选择最适合的算法。

结束语 本文针对视频点播系统，对目前存在的各种缓存算法做了比较详细的介绍、分析和总结；同时，还具体给出了各算法的实现方式，并通过使用模拟的数据和实际数据观察比较它们的不同；最后，对不同的缓存算法作出了总结分析。在视频点播环境下，LFU、LRFU和SC都是效率较高的算法，考虑服务器的内存大小、运行效率等，可选择对应的性价比最高的算法，并调整相应的参数，以此获得理想的效果。

参考文献

[1] Johnson T, Shasha D. 2Q: A low overhead high performance buffer management replacement algorithm[C]// Proceedings of VLDB Conf.. 1994:297-306

[2] Jiang S, Zhang X. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance[J]. ACM SIGMETRICS Performance Evaluation Review, 2002, 30(1): 31-42

[3] Lee D, Choi J, Kim J H, et al. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies[J]. IEEE transactions on Computers, 2001, 50(12): 1352-1361

[4] O'neil E J, O'neil P E, Weikum G. The LRU-K page replacement algorithm for database disk buffering[J]. ACM SIGMOD Record., ACM, 1993, 22(2): 297-306

[5] Duong N, Cammarota R, Zhao D, et al. SCORE: A Score-Based Memory Cache Replacement Policy[C]// JWAC 2010-1st JILP worshop on computer architecture competitions. 2010

[6] 魏维，罗时爱，刘凤玉. 视频点播中视频服务器节目替换算法研究[J]. 计算机工程与应用，2008，44(2)：245-248