一种针对邮件服务类应用改进的2Q*算法及其在存储缓存中的应用

孟晓烜[1,2] 司成祥[1,2] 刘振晗[1,2] 许 鲁[1]
(中国科学院计算技术研究所 北京 100080)[1] (中国科学院研究生院 北京 100039)[2]

摘 要 针对2Q算法对于邮件服务类负载所表现出的缓存性能特点提出了一种改进算法2Q*。模拟实验数据显示,改进后的2Q*算法在各种缓存容量下都优于包括经典2Q算法在内的其他替换算法。为了验证2Q*算法在真实系统中的有效性,将该算法集成于FlexiCache系统中并与目前主流的顺序自适应预取策略有机结合。实验结果表明,2Q*算法不仅能够在实际缓存系统中有效改善邮件服务类应用的物理I/O性能,而且其实际运行开销也非常低。

关键词 缓存,邮件服务,替换算法,2Q*
中图法分类号 TP333 **文献标识码** A

Replacement Algorithm Improved on 2Q* for Mail Service Workload and its Application in Storage Cache

MENG Xiao-xuan[1,2] SI Cheng-xiang[1,2] LIU Zhen-han[1,2] XU Lu[1]
(Institute of Computing Technologies, Chinese Academy of Sciences, Beijing 100080, China)[1]
(Graduate School of the Chinese Academy of Sciences, Beijing 100039, China)[2]

Abstract This paper analyzed the performance characteristics of classic 2Q algorithm when it was performed on mail-service workloads, and proposed an improved algorithm, called 2Q*. The simulation results show that 2Q* algorithm can outperform the other replacement algorithms, including the classic 2Q algorithm, for all the cache sizes and various mail-service workloads. To verify the simulation results in real practice, we implemented the algorithm in FlexiCache, a partitioned buffer cache system, and integrated it with a popular adaptive sequential prefect policy properly. The experiment results in real system further confirm the effectiveness of 2Q* algorithm for mail service kind of applications in improving their physical I/O performance. Moreover, its runtime overhead is also fairly low.

Keywords Buffer cache, Mail service, Replacement algorithm, 2Q*

1 引言

随着网络和存储技术的发展,集中式存储模式正在逐步取代传统的本地直连式存储模式。其中前者的优点在于:它不仅能够有效降低存储管理成本[1],同时便于实现应用间的数据共享、在线数据备份以及存储虚拟化等高级存储管理功能。在集中式存储模式下,存储系统作为一种共享资源通常需要同时服务于多种不同类型的应用。缓存作为一种重用的性能优化手段被广泛应用于各级存储系统来提高应用的I/O性能。传统缓存系统在设计上采用基于全局替换方法,因而不能够很好地适用于集中式存储模式下的缓存管理,这是由于共享缓存的多种应用往往具有不同类型的访问模式,单一的缓存替换算法会导致缓存资源利用率的低下[2]。为了解决该问题,我们在前期工作中提出了一种基于动态分区技术的缓存管理架构-FlexiCache[3-5]。在该架构下,每个应用具有相对独立的缓存分区。根据具体应用的负载访问模式,每个缓存分区可以配置更适合的替换算法,从而有效地提高缓存资源利用率。称该方法为基于替换算法的局部优化。

为了达成局部优化效果,我们需要分析替换算法对于各种不同类型应用负载的适用性。本文研究适用于邮件服务类负载[4]的替换算法。已知一个优秀的替换算法的重要标志是能够在中等缓存容量时取得相比于其他替换算法更好的性能[5]。基于此,在对已有替换算法的模拟实验中发现2Q算法[6]相比于其他算法更适合于邮件服务类负载[7]。如图2(a)所示,2Q算法对于邮件服务负载的命中曲线随缓存容量增长稳步提升且在中等缓存容量区间明显优于LRU算法。然而当缓存容量足够大时,2Q算法的性能反略低于LRU算法。为了能够在各种缓存容量配置下有效优化邮件服务类应用的物理I/O性能,本文有针对性地改进了2Q算法。改进后的2Q*算法不仅可以在中、小缓存容量时优于其他替换算法,同时能够有效改善经典2Q算法在大缓存容量下的性能。为了验证2Q*算法在真实系统中的有效性,将该算法应用于FlexiCache存储缓存系统中并与目前主流的顺序自适应预取策略有机结合[8]。实验结果表明,2Q*算法能够有效地改善

到稿日期:2009-04-17 返修日期:2009-07-24 本文受973国家重点基础研究发展计划(2004CB318205),863国家重点基金项目(2007AA01Z402,2007AA01Z184,2009AA01Z139,2009AA01A403)资助。

孟晓烜(1980—),男,博士生,主要研究方向为虚拟存储、共享缓存技术等,E-mail:xiao.x.meng@gmail.com;司成祥(1982—),男,博士生,主要研究方向为冗余数据保护技术等;刘振晗(1979—),男,博士生,主要研究方向为分布式文件系统等;许 鲁(1962—),男,博士生导师,主要研究方向为网络存储和操作系统等。

邮件服务类应用的 I/O 性能。

本文第 2 节分析了 2Q 算法对于邮件服务类负载所表现出的性能特点；第 3 节针对邮件服务类应用提出了改进的算法 2Q* 并给出相应的模拟实验结果；第 4 节讨论如何将 2Q* 算法应用于 FexiCache 系统；第 5 节给出 2Q* 算法在真实系统中的性能评测；最后总结全文。

2 性能分析

本节分析 2Q 算法对于邮件服务类负载所表现出的性能特点。分析的目的在于：1)指出 2Q 算法在中等缓存容量时的性能优势；2)指出导致 2Q 算法在大缓存容量时性能劣势的原因。为了便于分析，采用模拟实验方法并选择 LRU 算法作为 2Q 算法的性能对比对象，其中 LRU 算法是目前实际存储系统中普遍采用的一种替换算法，其最大的优点在于简单高效。本文实验中缓存块的粒度均设为 4kB，且使用的邮件服务类负载(负载统计信息参如表 1 所列)均采集于惠普公司邮件服务器的后端存储系统，其中邮件服务器的管理软件版本为 Openmail。在本节模拟实验中，选用第 5 号邮件服务器负载进行分析。为了便于讨论，依据邮件服务负载的数据集大小将缓存容量依次划分为以下 3 个区间：小容量[32～256MB]、中容量[0.25～1.25GB]和大容量[1.25～2GB]。下面首先分析邮件服务类负载的特点。

表 1 3 种邮件服务类负载的统计信息

负载	存储容量	时长	请求数	读比例%	平均请求长度
openmail1	360GB	1h	2.58M	45.40	8kB
openmail5	342GB	1h	2.58M	61.34	8kB
openmail6	445GB	1h	3.13M	62.32	8kB

2.1 负载特点

图 1(a)给出了邮件服务负载的重复访问频率分布，即在不同访问频度下数据块再次被访问的概率分布；图 1(b)给出了数据块在不同访问频度下被再次访问的平均相邻访问间距(IRG inter-reference gap)分布；图 1(c)则给出了邮件服务负载中数据访问落在不同相邻访问间距内的柱状图分布，图中 x 轴的每一坐标点 x_i 代表 IRG 落在区间$[x_i, x_{i-1}]$内的数据访问总和。从中可以归纳出邮件服务负载的以下 3 个特点：1)由图 1(a)可知，负载中存在相当数量的非活跃数据，据统计仅访问 1,2,3,4 和 5 次的数据块所占比例依次分别约为 15%,21%,10%,13%和 8%，而重复访问超过 5 次的数据块则仅占35%；2)随着数据访问频度的增大，平均相邻访问间距随之显著下降，观察图 1(a)可知绝大多数的数据块在前 5 次访问中的 IRG 较大，此后则相对较小；3)由图 1(c)所示，绝大多数的数据访问 IRG 落在 IRG 分布区间两端，仅有非常少量的数据访问 IRG 处于 IRG 分布的中间区间。

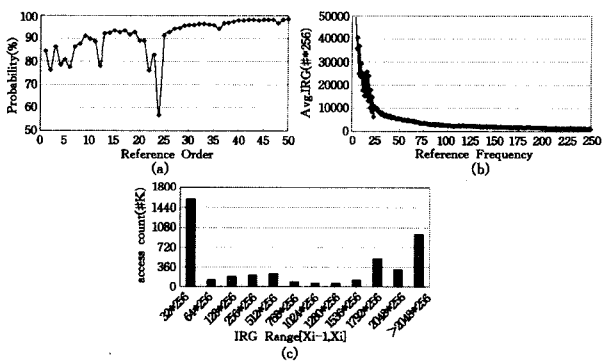

图 1 (a)邮件服务负载中重复访问概率分布、(b)平均相邻访问间距随访问频度变化分布、(c)数据访问在不同相邻访问间距区间内的分布

2.2 算法对比

基于邮件服务负载的上述 3 个特点，在下一小节中对比分析 2Q 算法和 LRU 算法的性能差异。本文实验中 2Q 算法的 q1. in 队列长度限制为缓存容量的 1/10；而 q1. out 队列大小则与真实缓存容量相当。图 2(a)给出了邮件服务负载在两种替换算法下的命中率变化曲线，其中测试的缓存容量从 32MB 变化至 2GB。由图可知，2Q 算法在中小缓存容量范围内(32MB～1.25GB)的性能均优于 LRU 算法，特别是在缓存容量介于 256MB 和 1.25GB 之间时，2Q 算法的性能优势非常明显。当缓存容量为 0.5GB 时，2Q 算法的缓存命中率比 LRU 算法高出 10%。然而随着缓存容量的进一步增长，2Q 算法的性能反而不如 LRU 算法。当缓存容量为1.75GB 时，2Q 算法命中率比 LRU 算法低了 4%。下面将对上述两种算法表现出的特点做进一步分析，重点分析 2Q 算法在中等缓存容量时的性能优势及其在大缓存容量下相对性能下降的原因。

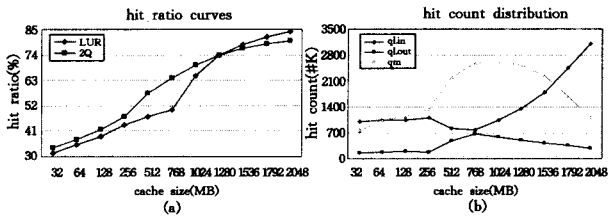

图 2 (a)邮件服务负载的缓存命中率曲线分布对比 (b)2Q 算法中 3 种不同队列的累积命中次数随缓存容量变化分布

2.2.1 2Q 算法

2Q 算法中 q1 队列用于识别负载中的活跃数据。q1 队列是按照 FIFO 方式进行管理的，一旦数据访问在 q1 队列中命中，则相应将缓存数据迁移至 qm 队列；qm 队列是以 LRU 的方式管理当前负载中识别出的活跃数据。这里非活跃数据是指相邻访问间距较大或者仅访问一次的数据块，反之活跃数据是频繁多次访问且相邻访问间距相对较小的数据。由于 q1 队列在功能上只是为了能够快速区分出负载中的非活跃数据，为了降低 q1 队列所占缓存容量比例，q1 队列被划分为 q1. in 和 q1. out 两部分。其中前者在缓存中，而后者只是记录数据块 ID 而并不保存数据内容本身。q1. in 队列长度限制(取决于 Klin 参数)通常设为缓存总容量的很小一部分，如 1/10[6,9]。此外，仅当数据在 q1. out 队列中命中时才被识别为活跃数据并迁移至 qm 队列首。而当数据在 q1. in 队列中命中时算法并不做任何处理，其原因在于这样可以避免将负载中仅在某一较短时段内被连续多次关联访问(correlated references)[10]的数据块识别为活跃数据。显然，短时内对同一数据块的连续多次关联访问并不能客观地反映数据当前的访问热度。图 2(b)给出了数据访问分别在 q1. in,q1. out 和 qm 3 个队列中的累积命中次数随缓存总容量变化的分布。在小缓存容量时，由于数据块的前几次访问的相邻访问间距较大(邮件服务负载特性 2)，因此数据块在前几次访问时不仅被快速地从 q1. in 队列中替换出去，同时很难在 q1. out 队列中命中。然而随着数据访问频度的增加，数据块的相邻访

问间距显著下降(邮件服务负载特性 2),因此数据块在高频访问时在 q1. in 和 q1. out 队列中命中的几率相应明显增大。又由于负载中访问频度较高的数据块所占比例非常小(邮件服务负载特性 1),因此在小缓存容量时 q1. out 队列中的命中次数也非常有限。然而,就是这些少量从 q1. out 队列中识别出的活跃数据可以长期缓存在 qm 队列中,从而在一定程度上提高了缓存命中率。这也正是 2Q 算法在小缓存容量时略优于 LRU 算法的原因。

在中等缓存容量区间时,如图 2(b)所示,随着缓存容量的增长,q1. out 队列的命中次数持续增长,相应的 qm 队列中的缓存命中次数也相应增长,而 q1. in 队列的命中次数则相反随之下降。这是相当数量的活跃数据在前几次访问时在 q1. out 队列中命中而直接加入 qm 队列所致。由图可知,q1. in 和 qm 两队列的累积命中次数总和则呈稳步上升趋势。

在大缓存容量区间时,一方面随着缓存容量的增长,q1. in 队列的命中次数随缓存容量的增长显著提高,而 q1. out 队列的命中次数反而呈急剧下降的趋势,这是当缓存容量足够大时初始数据访问在 q1. in 队列中的命中几率显著增大所致;另一方面,从 q1. out 队列中识别出的活跃数据的减少直接导致了 qm 队列命中次数的相应下降。由邮件服务负载特性 2)可知,后续数据访问的 IRG 随访问频度的增加而变得越来越小。而 q1. in 队列的 FIFO 管理方式意味着任何数据为了能够长期驻留于缓存中,必须在 q1. out 队列中经历一次缓存失效,我们称这种失效为强制识别失效(compulsory identification miss)。显然,在大缓存容量区间仍对 q1. in 队列采用 FIFO 管理方式是没有必要的。此外,额外的强制识别失效开销会带来 2Q 算法缓存性能的相对下降。

2.2.2 LRU 算法

从图 2(a)可以看出,LRU 缓存命中率曲线在中等缓存容量时的上升速率要明显低于 2Q 算法。经分析发现,这是邮件负载特性 3)所致,即数据访问 IRG 集中在 IRG 分布的两端,因而在处于中等缓存容量时,相当数量的数据访问 IRG 始终高于当前的缓存容量。这使得 LRU 算法无法有效利用负载中的访问局部性,从而导致缓存命中率随缓存容量增大其提升相对很小。相比之下,2Q 算法能够利用 q1 队列快速有效地区分出当前负载中活跃和非活跃数据,并将识别出的活跃数据缓存在独立于 q1. in 队列的 qm 队列中,从而有效提高后者的缓存空间利用率。正因如此,2Q 算法的缓存命中率曲线可以在中等缓存容量区间时仍随缓存容量的增长而稳步增长。

2.2.3 性能分析小结

由以上分析,可以得出以下结论:

①当缓存容量相对有限时(中、小缓存容量区间),2Q 算法通过区分负载中的活跃和非活跃数据来有效提高缓存空间利用率。相反,LRU 算法由于邮件服务负载特性而不能够高效地利用缓存资源,特别当处于中等缓存容量时,LRU 算法的命中率曲线随缓存容量的增大而上升得相对缓慢。

②当缓存容量相对充足时(大缓存容量区间),数据访问在 2Q 算法中的 q1. in 队列命中机率增大,因而其 FIFO 的管理方式不仅不能够有效利用负载中局部性,反而会带来了额外的强制识别失效开销。这正是 2Q 算法的命中率曲线在大缓存容量区间时反被 LRU 算法曲线超出的原因。

3 2Q 算法的改进——2Q*

3.1 基本原理

通过 3.1 节的分析可知 2Q 算法中 q1. in 队列中的 FIFO 管理方式是直接导致其在大缓存容量时性能相对下降的原因。为了能够有效降低 2Q 算法在大缓存容量时由于强制识别失效所带来的不必要的性能损失,我们对 2Q 算法中的 q1 队列管理方式进行相应的改进:采用 LRU 方式管理 q1. in 队列,而沿用原先的 FIFO 方式管理 q1. out 队列,我们称改进后的算法为 2Q*。这样改进的依据在于:1) q1. in 队列管理方式在改为 LRU 方式后,2Q 算法在大缓存容量下的行为与 LRU 算法相似。具体来说,在缓存预热阶段首次失效数据都是加入到 q1. in 队列中。当缓存容量足够大时,绝大多数的数据访问都可在 q1. in 队列中命中,且又因为 q1. in 队列是按照 LRU 方式管理的,因此在 q1. in 队列中命中的缓存数据又被重新移入 q1. in 队列头,从而仅有很少量的数据访问会在 q1. in 队列中失效而在 q1. out 中命中。换言之,在大缓存容量下,2Q* 算法会将绝大多数缓存容量分配给按照 LRU 方式管理的 q1. in 队列。在 3.3 节中将通过模拟实验数据来验证这一点。已知当缓存容量足够大以至于和负载中活跃数据访问集的大小相当时,LRU 算法被公认为是最有效的替换算法[9],因此改进后的 2Q* 算法可以有效改善 2Q 算法在大缓存容量下的性能。2) 通过严格限制 q1. in 队列的长度(如在本文实验中 K1in 参数均设为 1/10),改进后的 2Q* 算法对于 q1 队列的不同管理方式并不会影响其在缓存容量有限时对于负载中活跃数据识别的有效性,即能够保证经典 2Q 算法在中、小缓存容量时的性能优势。同样在 3.3 节中将通过模拟实验数据来验证这一点。

3.2 算法描述

图 3 给出了 2Q* 算法的伪代码描述。下面分别阐述该算法对于失效、命中和替换等 3 种不同缓存管理操作的处理流程:

```
access(block b) {
    if b is in qm {
        move b to the head of qm
    } else if b is in q1.in {
        move b to the head of q1.in
    } else {
        if b is in q1.out {
            remove b.identifier from q1.out
            evictfor(b)
            add b to the head of qm
        } else {
            evictfor(b)
            add b to the head of q1.in
    }
}

evictfor(block x) {
    if cache is not full {
        put x into a free block slot
    } else {
        if size of q1.in larger than Kin {
            evict the tail of q1.in, call it y
            if q1.out is full {
                remove identifer of z from the tail of q1.out
            }
            put the identifier of y to the head of q1.out
        } else {
            evict the tail of qm, call it z
        }
        put x into the evicted block slot
    }
}
```

图 3 2Q* 算法流程的伪码描述

1) 缓存命中处理

1-1) 当被访问数据块在 q1. in 队列中命中,数据块将被移至 q1. in 队列首;

1-2) 当被访问数据块在 qm 队列中命中,数据将被移至 qm 队列首。

2) 缓存失效处理

2-1) 当被访问数据块在 q1. out 队列中命中,数据块作为当前识别出的活跃数据而被加入 qm 队列首,该数据块在 q1. out 队列中的 ID 也将相应被删除;

2-2) 当被访问数据块没有在 q1. out 队列中命中,数据块作为当前非活跃数据而被加入 q1. in 队列首。

3) 缓存替换处理

3-1) 当前缓存处于预热阶段(即仍有空闲缓存块),那么从空闲缓存中挑选出一块来存储新近失效数据块内容;

3-2) 当前缓存处于饱和阶段(即已无空闲缓存块)且 q1. in 队列的长度超过 *Klin* 参数限制,那么替换出当前处于 q1. in 队列尾部的数据块来存储新近失效数据块内容;

3-3) 当前缓存处于饱和阶段且 q1. in 队列的长度不足 *Klin* 参数限制,那么替换出当前处于 qm 队列尾部的数据块来存储新近失效数据块内容。

3.3 模拟实验结果

本小节通过模拟实验对比改进后的 2Q* 算法与已有替换算法的性能并重点分析导致 2Q* 算法与经典 2Q 算法之间性能差异的原因。实验结果如图 4 所示,改进后的 2Q* 算法在中、小缓存容量区间内的性能与 2Q 算法相当,优于包括 LRU,LFU,ARC[11] 和 LIRS[12] 在内的 4 种经典替换算法。以第 5 号邮件服务器负载为例,2Q* 算法在中等缓存容量下提升缓存命中率最高可达 30%(相对于缓存容量为 0.75GB 时的 LRU 算法);而在大缓存容量时,2Q* 算法与 LRU 算法相当,略优于包括经典 2Q 算法在内的其他 4 种替换算法。

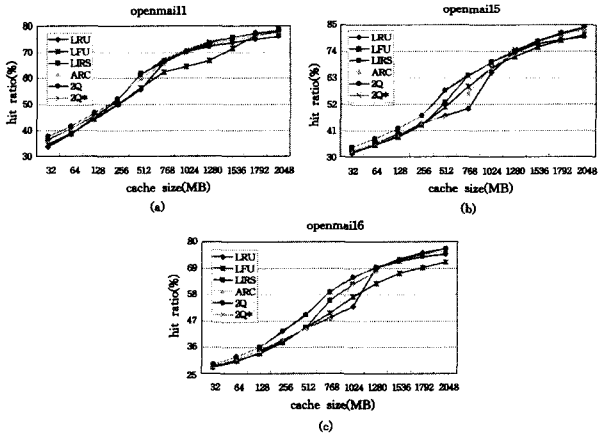

图 4 多种邮件服务负载在各种不同替换算法下的缓存命中率曲线对比

下面以第 5 号邮件服务器负载为例进一步分析 2Q* 与 2Q 算法的性能差异缘由。图 5(a-c)分别给出了改进前后的 2Q 算法中 q1. in,q1. out 以及 qm 等 3 队列的累计命中次数随缓存容量的分布对比。从图中观察可得以下几点:1) 2Q* 算法提高了数据访问在 q1. in 队列中的命中次数,且随着缓存容量的增长改进前后两种算法在 q1. in 队列中的命中次数差距不断被扩大,显然这是由于 2Q* 算法在采用 LRU 方式后使得在大缓存容量时更多的数据访问在 q1. in 队列中命中;2) 2Q* 算法使得失效数据访问在 q1. out 队列中的命中次数在大缓存容量时明显降低,这是由于在大缓存容量时,改进后的 2Q* 算法使得更多的数据访问在 q1. in 队列中命中,从而也就相应降低了失效数据访问在 q1. out 队列中的命中几率;3) 2Q 算法中的 qm 队列命中次数要明显高于 2Q* 算法,且随着缓存容量的增大两者之间的差距在逐步加大,这是由于在 2Q* 算法中所识别出的活跃数据量相对于经典 2Q 算法要少,显然活跃数据量的减少必然带来了 qm 队列中数据访问命中次数的下降。但是通过对比图 5(a)和图 5(c),可以发现 2Q* 算法对于 q1. in 队列命中次数的提升在绝对值上要明显高于其对于 qm 队列命中次数的降低,此差异的原因正如在 3.1 节中所预期的那样:2Q* 算法将 q1. in 队列的管理方式由原先的 FIFO 方式改变为 LRU 方式后不仅没有影响该算法在中、小缓存容量时对于活跃数据识别的有效性,而且在大缓存容量时有效降低了由强制识别失效所带来的缓存性能开销。如图 4(b)所示,2Q* 算法在缓存容量为 1.75GB 时较经典 2Q 算法缓存命中率提升约 5%。此外,由图 5(b)和图 5(c)可以推断在大缓存容量时,2Q* 算法中的缓存数据大都聚集于 q1. in 队列中。显然,在缓存容量与负载活跃数据集大小相当时[9],LRU 是达成缓存效率最高的工作方式。

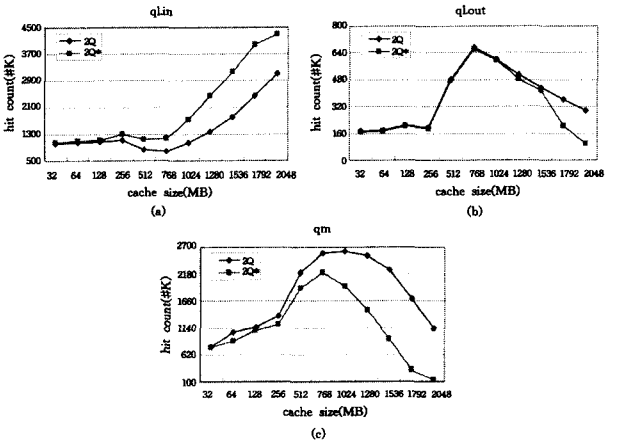

图 5 改进前后 2Q 算法中 q1. in,q1. out 以及 qm 等 3 队列的累计命中次数随缓存容量变化的分布对比

4 在存储缓存中的应用

为了验证 2Q* 在真实系统中的有效性,我们将它实现于 Flexicache 系统中。FlexiCache 是一种针对存储服务器设计的动态分区缓存管理系统[3-5],如第 1 节所述,它以应用为单位将缓存资源划分为不同的分区,根据具体应用负载特征,不同的缓存分区可以配置不同的局部替换算法,以提高分区缓存资源利用率。为了便于在 FlexiCache 中集成各种不同的替换算法,该系统在机制上提供了一种标准化的通用分区缓存管理接口。这样,为了在 FlexiCache 系统中实现新的替换算法,只需要实现分区缓存管理接口中定义的各种缓存管理方法。然而具体替换算法中所使用的各种独特的实现机制(如缓存元数据管理等)对于 FlexiCache 中其他子系统模块是完全透明的。这种模块化的设计方法不仅简化了替换算法在 FlexiCache 中的实现,同时有助于提高 FlexiCache 系统代码的可维护性和可扩展性[3]。

4.1 缓存管理模型

目前 FlexiCache 系统中的分区缓存管理接口定义了 3 种核心缓存管理方法。下面简要介绍这 3 种方法所实现的缓存管理功能:

Cache_Insertion 方法将失效数据块加入到应用缓存分区中;

Cache_Renewal 更新命中数据块在应用缓存分区中的状态;

Cache_Eviction 从应用缓存分区中替换出指定数量的空闲缓存块。

上述 3 种核心缓存管理方法实际上也就定义了 Flexi-

Cache 系统中应用分区的缓存管理模型[3]。具体来说，当缓存失效时，从系统预留的空闲缓存池中分配所需的缓存块；而当缓存池中的空闲缓存块数低于一定阈值时，系统则相应地从各个应用分区中分别回收一定数量的空闲缓存块。各应用分区中具体应替换出哪些缓存块，取决于负责管理该应用分区的局部替换算法。由此可知，Cache_Insertion 和 Cache_Renewal 两方法都是在应用 I/O 线程的上下文中调用并分别用于处理缓存失效和缓存命中两种情形；而 Cache_Eviction 方法是在系统缓存资源回收线程的上下文中调用[3]。FlexiCache 中的缓存管理模型与传统替换算法设计时所基于的缓存管理模型之区别在于以下两点：1）缓存替换操作与缓存失效处理的分离；2）应用分区缓存容量的动态变化。为了将 2Q* 算法集成于 FlexiCache 系统中，需要对算法本身的处理逻辑做一定的修改，并相应实现上述 3 种核心缓存管理方法。图 6 以伪码形式概要性地给出了 2Q* 算法在 FlexiCache 系统中的实现（这里忽略了实际系统实现时所作的优化，如回写和缓存数据同步等），其基本处理流程如下（注：本小节暂不讨论与预取策略相关的处理逻辑）：在 2Q* _Insertion 方法中，2Q* 算法将在 q1. out 队列中命中的失效数据加入 qm 队列中，并标记为活跃数据块，反之加入 q1. in 队列中。如图 6 所示，我们用 *x. hot* 标记 x 块为活跃数据；在 2Q* _Renewal 方法中，2Q* 算法将命中数据块重新移至所在队列首；而在 2Q* _Eviction 方法中，2Q* 算法首先根据应用分区在回收操作完成后的实际缓存容量（$tgrtblk = curblk - count$）和算法预先设置的 $Klin$ 参数值来调整 q1. in 队列长度限制（$limit_{q1.in} = tgrtblk * Klin$），接着根据当前 q1. in 和 qm 两队列长度选择应从哪个队列进行回收，最后从队尾开始遍历所选择的回收队列并从中回收指定数目的缓存块。

```
2Q*_insert(block x) {
    if x is in q1.out {
        remove x from q1.out
        set x.hot to TRUE
        if x.pref is TRUE {
            add x to head of q1.in
        } else {
            add x to head of qm
        }
    } else {
        add x to the head of q1.in
    }
}

2Q*_renewal(block x) {
    if x.pref is TRUR {
        set x.pref to FALSE
    }
    if x.hot is TRUE {
        move x to head of qm
    } else {
        move x to head of q1.in
    }
}

2Q*_evict(count) {
    get tgrtblk by subtract count from curblk
    set limit_q1.in to tgrtblk*Klin
    if |q1.in| larger than limit_q1.in {
        set evict_list to q1.in
    } else {
        set evict_list to qm
    }
    for each scaned block y do //scan evict_list from head
        to tail removey from evict_list
        if evict_list is q1.in {
            add y.identifier to the head of Q_out
        }
        decrement count by one
        if count equal zero {
            jump out of scan loop
        }
    done
}
```

图 6　2Q* 算法在 FlexiCache 系统中的伪码实现描述

4.2　预取策略结合

在实际缓存系统中，替换算法通常与预取策略相结合，以优化应用的 I/O 性能。其中替换算法可以利用 I/O 负载中的局部性原理来提高缓存效率，而预取则可以通过利用 I/O 负载中的相邻请求间隙或思考时间（think time）来屏蔽慢速磁盘 I/O 对应用 I/O 性能的影响。在 FlexiCache 系统中，我们采用默认的自适应线性预取策略，该策略不仅能够在线探测缓存负载中潜在的顺序访问模式，同时能够自适应地调节预取的粒度和预取的时间。关于该策略的介绍可参考文献[8]，这里仅讨论 2Q* 算法如何在 FlexiCache 系统中与预取策略有机结合以进一步提高应用的 I/O 性能。如图 6 所示，在 2Q* _Insertion 方法中，2Q* 算法将预取数据不加区分（不考虑其是否活跃数据）地统一加入至 q1. in 队列中，它通过增加 *x. pref* 标记来记录 x 块当前是否为预取数据。且此时应用分区中存在两类缓存数据，其中一类是应用同步访问数据，另一部分为预取策略异步预取数据。在 2Q* _renewal 方法中，如果当前缓存命中数据块 x 为预取数据，那么 2Q* 算法需要判断其是否为活跃数据，即 *x. hot* 标记是否为真。如果是，那么数据块 x 将被移至 qm 队列首，反之亦然。

5　性能评测

本节对 2Q* 算法在 FlexiCache 中的实现进行性能评测。其中 5. 1 节对比 2Q* 算法和包括经典 2Q 在内的其他 3 种替换算法在真实存储系统中的性能；5. 2 节通过对比配置预取策略前后 2Q* 算法的性能差异来验证 2Q* 算法是否与预取策略有机地结合，以进一步提升应用的 I/O 性能；5. 3 节通过对比 2Q* 算法和 LRU 算法以及 Linux VFS 缓存在顺序读写模式下的聚合带宽差异，来检验 2Q* 算法在实际系统中的运行开销。在讨论上述 3 种性能评测结果之前，首先给出测试所基于的硬软件实验环境。

测试中使用的存储服务器的硬软件配置如表 2 所列，其中 4 块西捷 250GB 的 SATA 磁盘通过 3WareRAID 控制器按每两个一组以 RAID0 方式构成两个存储卷。使用的测试工具包括 xdd 和 kernio 两种，其中前者是一种运行于用户态的磁盘测试工具[13]，后者为运行于内核空间的块级 I/O 负载回放器[3]，它能够统计诸如平均响应延迟、每秒操作数和磁盘负载等多种 I/O 性能参数指标。

表 2　存储服务器硬软件配置

存储服务器	Gentoo-2. 6. 18Kernel Intel(R) Xeon(TM) 3. 00GHz CPU 2GB DDR Memory 3ware-9500RAID Card RAID0：4×WD2500SD(SATA，250GB，7200RPM)

5.1　算法性能对比

本小节通过对比邮件服务负载在 FlexiCache 系统中的不同替换算法配置下实测得到的平均响应延迟来验证 2Q* 算法在实际存储系统中的有效性。作为比较，在 FlexiCache 系统中实现了 LRU，2Q，ARC 和 2Q* 等 4 种替换算法。测试中利用 kernio 工具通过 FlexiCache 缓存向物理存储卷回放邮件服务负载并统计回放得到的平均响应延迟。图 7 给出了第 5 号邮件服务器的实验结果（注：其他负载的实验结果相似。此外由于受硬件资源所限，测试的最大缓存容量为 1. 25GB，但这并不影响性能分析的有效性）。如图所示，2Q* 算法在各种缓存容量下均优于包括经典 2Q 算法在内的其他 3 种替换算法，该算法可降低负载平均响应延迟最大约达 20%（如 1GB 缓存容量时的 ARC 和 2Q 算法，0. 75GB 容量时的 LRU 算法）。实际系统中的实验结果进一步验证了 2Q* 算法在模拟实验中对于邮件服务负载所表现出的良好性能。

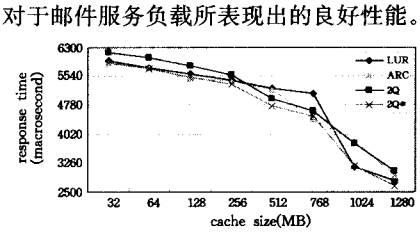

图 7　各种替换算法下的平均响应延迟随缓存容量变化分布

5.2 预取效果评测

本小节测试的目的在于检验自适应顺序预取策略对于2Q* 算法性能的影响，即 2Q* 算法是否能够有效地与顺序预取策略相结合以进一步提升应用的 I/O 性能。由于本文测试中使用的磁盘子系统性能低于负载采集系统中的磁盘子系统性能，因此我们将相邻请求间的时隙扩大原始负载中的 4 倍，这样可有利于发挥预取策略对于慢速磁盘 I/O 延迟的屏蔽作用。图 8 给出了第 5 号邮件服务器负载的实验结果(注：其他负载的实验结果类似)。其中图 8(a)对比了 2Q* 算法在有无预取两种情形下的 I/O 性能。如图所示，在结合预取策略后，负载平均响应延迟在各种缓存容量下均得到了降低，约为 5%左右。图 8(b)相应给出了预取窗口大小随时间变化的分布。由图可知，在负载回放过程中预取窗口大小稳定在 32kB～64kB 左右(相当于 8～16 个缓存块)。而由表 1 可知，邮件服务负载的平均请求大小仅为 8kB 左右。由此可知，2Q* 算法在结合预取策略后通过提前从磁盘子系统中读取当前已探测到的顺序访问序列并利用负载中存在的请求间隙，来尽可能屏蔽慢速磁盘 I/O 延迟对于负载回放性能的影响。实验数据表明，2Q* 算法在与预取策略有机结合后可以进一步优化邮件服务类应用的 I/O 性能。

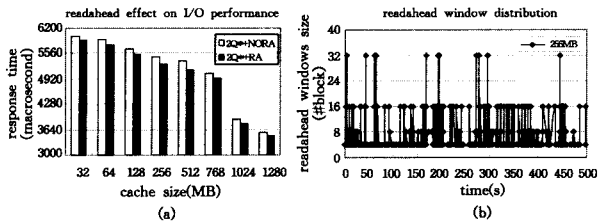

图 8 自适应顺序预取策略对于 2Q* 算法性能的影响

5.3 算法开销评测

本小节测试的目的在于检测 2Q* 算法在实际系统中的运行开销。低开销是任何一种可实用化缓存技术必备的特点之一。我们选用 LRU，ARC 算法以及 Linux 的 VFS 缓存(注：VFS 缓存采用一种基于 Clock 方式的类 2Q 替换算法[14])作为性能评测对象，其中前两者可以对比 2Q* 与其他替换算法在同一缓存系统平台下的运行开销，后者可以用于对比 FlexiCache 在采用了 2Q* 算法后与目前主流高性能缓存系统之间的运行开销。为了避免替换算法对于 I/O 性能的影响，我们利用 xdd 工具同时顺序读写两个存储卷并分别测试 FlexiCache 和 VFS 缓存所能取得的聚合带宽峰值。测试中缓存总容量均设为 256MB，数据读写量均为 2GB，读写粒度从 4kB 至 1MB 不等。实验结果如图 9 所示。可见，1) 首先在 FlexiCache 系统下 2Q* 算法的开销与 LRU 和 ARC 相当；2) FlexiCache 系统的聚合写性能在包括 2Q* 算法在内的各种替换算法配置及各种读写粒度下均远高于 VFS 缓存。经 top 工具分析发现，这是由于在聚合写测试中，CPU 成为了性能瓶颈。因为为了尽快回收新的空闲缓存块，回收线程需要不停地遍历缓存链表，以找到干净的数据块进行替换。VFS 缓存由于所采用的回写和替换机制相对于 FlexiCache 更为复杂，因而在测试中更快地达到了 CPU 饱和点。本小节的实验数据表明，2Q* 算法在实际缓存系统中的开销足以满足实用化的要求。

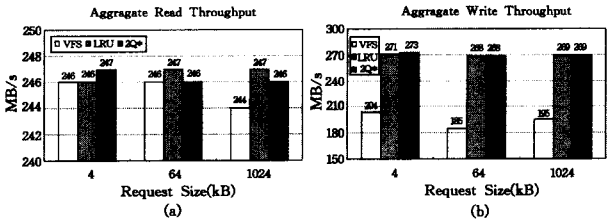

图 9 各种替换算法在顺序 I/O 模式下的聚合读写带宽

结束语 本文针对 2Q 算法对于邮件服务类负载表现出的缓存性能特点提出了一种改进算法 2Q*，它将经典 2Q 算法中 q1. in 队列的 FIFO 管理方式改为 LRU 方式。模拟实验数据表明，改进后的算法 2Q* 在保持经典 2Q 算法在中、小缓存容量下相对于其他替换算法的性能优势的同时，还能够有效地解决经典 2Q 算法在大缓存容量时性能相对下降的问题。为了验证模拟实验结果，将 2Q* 算法集成于基于分区技术的 FlexiCache 缓存系统中，并与目前主流的自适应顺序预取策略有机结合，以进一步优化应用在实际系统中的 I/O 性能。实验结果表明，2Q* 算法不仅可以有效改善邮件服务类应用的物理 I/O 性能，同时其实际运行开销也非常小。

参 考 文 献

[1] 那文武，孟晓烜，柯剑，等. BW-VSDS：大容量、可扩展、高性能和高可靠性的网络虚拟存储系统[C]// 中国计算机大会. 2008

[2] Konanki P, Butt A R. FlexiCache: A Flexible Interface for Customizing Linux File System Buffer Cache Replacement Policies [C]// Proceedings of USENIX conference on File and Storage Technologies. 2007

[3] Meng X X, Si C X, Na W W, et al. A Flexible Two Layer Buffer Caching Scheme Designed for Storage Cache[C]// Proceedings of IEEE International Conference on High Performance Computing and Communications. 2009

[4] 孟晓烜，那文武，徐伟，等. 一种针对存储服务器设计的动态分区缓存管理系统[J]. 计算机研究与发展，2009(已接收)

[5] 孟晓烜，李一鸣，卜庆忠，等. 一种面向存储服务的缓存管理模型[J]. 计算机工程，2009，35(1)：30-33

[6] Johnson T, Shasha D. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm[C]// Proceedings of Very Large Databases Conference. 1995

[7] http://tesla.hpl.hp.com/public_software/

[8] Gill B S, Bathen L A D. AMP: Adaptive Multi-stream Prefetching in a Shared Cache[C]// Proceedings of USENIX Conference on File and Storage Technologies. 2007

[9] Butt A R, Gniady C, Hu Y C. The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms[J]. IEEE Transactions on Computers, 2007, 56(7): 889-908

[10] O'Neil E, O' Neil P, Weikum G. The LRU-K page Replacement Algorithm for Database Disk Buffering[C]// Proceedings of ACM SIGMOD Conference. 1993

[11] Megiddo N, Modha D S. ARC: A Self-tuning, Low Overhead Replacement Cache[C]// Proceedings of USENIX Conference on File and Storage Technologies. 2003

[12] Jiang S, Zhang X. LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance[C]// Proceedings of SIGMETRICS Conference. 2002

[13] http://www.ioperformance.com/

[14] Bovet D P, Cesati M. Understanding Linux kernel[M]. America: O'Reilly Press, 2005