中国科学: 信息科学 2011 年 第 41 卷 第 4 期: 429–439
www.scichina.com info.scichina.com
《中国科学》杂志社
SCIENCE CHINA PRESS

论 文

基于顺序检测的双队列缓存替换算法

肖侬*, 赵英杰*, 刘芳, 陈志广

国防科学技术大学计算机学院, 长沙 410073
* 通信作者. E-mail: nongxiao@nudt.edu.cn, hyperseymour@163.com

收稿日期: 2009–12–08; 接受日期: 2010–04–28
国家自然科学基金 (批准号: 60736013, 60903040, 61025009)、教育部新世纪优秀人才支持计划 (批准号: NCET08-0145) 和国家高技术研究发展计划 (批准号: 863-2006AA01A106) 资助项目

摘要 缓存技术是提高存储性能最有效的技术之一, 在存储系统中得到了广泛应用. 由于缓存容量有限, 替换算法在缓存策略中占据了重要地位. 当前, 缓存替换算法的研究工作主要集中在如何提高缓存系统命中率, 忽略了通过降低缓存失效开销来提高缓存系统性能方面的研究. 针对这一问题, 本文提出了一种基于顺序检测的双队列缓存替换算法: 本算法优先淘汰缓存中的顺序页面, 保留随机页面, 从而大大减少后续请求对磁盘进行随机访问的次数, 能够显著降低缓存系统的失效开销. 同时, 本算法使用两个队列分别维护新加入页面和待淘汰页面, 遵循时间局部性原理, 保证了缓存命中率. 实验结果表明, 本算法在多种缓存大小及工作负载下, 可以达到比 LRU 和 ARC 算法更优的性能.

关键词 缓存技术 替换策略 命中率 失效开销 顺序检测

1 引言

计算机系统各组成部分的发展是非均衡的. 在过去的十年中, 内存的性能提高了不止一个数量级, 而同一时期内, 由于主体部分是机械设备, 磁盘的性能提高了只有大约 10%[1]. 如果机械磁盘技术没有突破, 磁盘与内存之间的性能差距将会进一步加大. 存储系统已经成为整个计算机系统的性能瓶颈, 尤其是难以满足数据密集型应用对高带宽、高数据传输率存储系统的巨大需求.

缓存技术是减少磁盘与内存性能差距, 提高存储系统整体性能的基础技术之一, 广泛应用于各种文件服务器、存储服务器、数据库服务器、流媒体服务器和网页服务器中. 由于内存价格比磁盘高很多, 通常存储系统配置的内存容量比磁盘小. 缓存的容量不足以存储所有有价值的页面, 当缓存中没有空闲空间时, 就必须选择价值较低的页面进行替换. 这种选择页面的依据就是替换策略, 它极大地影响了缓存系统的整体性能, 是缓存设计中的关键问题.

现有的缓存替换策略的研究主张以提高缓存命中率的方式改善缓存系统性能, 然而, 命中率能否提高是受实际工作负载所制约的. 1992 年, Muntz 和 Honeyman[2] 在一份研究报告中就指出, 存储服务器缓存命中率很低, 以提高缓存命中率为目标的算法在服务器中表现不佳. 这主要有 3 个原因. 首先, 尽管服务器常常配备大容量的缓存, 但是服务器安装有更大容量的硬盘, 缓存容量的增长远不及硬盘容量的增长, 所以只有很小一部分页面可以被缓存起来. 因此, 服务器缓存的命中率很低, 只有 10%–30% 左右. 其次, 服务器由多个客户端的多道程序共享使用, 大量并发访问引起的过度竞争以及

引用格式: 肖侬, 赵英杰, 刘芳, 等. 基于顺序检测的双队列缓存替换算法. 中国科学: 信息科学, 2011, 41: 429–439

工作负载的快速变化, 使得服务器访问的局部性很差, 这进一步导致了缓存命中率的下降. 最后, 对于多级缓存系统而言, 当客户端缓存失效时, 页面访问才有可能在服务器端命中. 随着客户端缓存容量逐年增长, 服务器端缓存命中率的提升潜力更加有限.

当命中率难以进一步提高时, 是否存在其他的方法可以改善缓存系统的整体性能? 我们可以从磁盘的访问特性得到启发. 事实上, 磁盘是非匀速访问的, 存在随机访问速率和持续访问速率两个指标. 随机访问速率在过去 10 年中只提高了约 10%, 而持续访问速率提高了十几倍, 图 1 对比了从 1997 到 2007 年 10 年间, 典型的 2.5 和 3.5 寸磁盘平均访问延迟 (反比于随机访问速率) 和持续访问速率的变化. 由图 1 可见, 持续访问速率与随机访问速率无论从绝对速率对比还是从增长速度对比都要快得多, 而且其速度差异随着技术发展呈现不断扩大之势. 以希捷公司的产品 Barracuda7200.11(ModelNumber: ST31000340AS) 为例, 其平均访问延迟为 12.98 ms, 持续传输速率为 105 MB/s. 如果每次请求平均访问 4 KB 数据, 持续访问速率是随机访问速率的 350 倍. 这种差异存在的根本原因在于: 数据定位所需要的磁头寻道和旋转操作是非常费时的, 而定位之后进行的单纯的数据读取操作的开销则相对小得多. 当缓存失效时, 首先需要在磁盘上定位失效页面, 然后读出并装载入内存. 所以缓存失效开销不是一个简单的常量, 而依赖于当前磁头的位置以及请求页面的位置, 因此间接地与请求页面以及其前驱后继请求的连续页面序列的长度有关, 这样就为降低缓存失效开销提供了很大的潜力以及可操作性. 本文正是从这一原理出发, 提出一种具有很低的失效开销的缓存替换算法, 它通过对请求页面进行地址分析, 优先淘汰比较连续的页面, 保留随机分布或者连续程度不大的页面, 这样缓存失效时会减少磁头的机械移动, 降低失效开销, 从而改善整个缓存系统的访问性能.

本文的研究工作具有如下贡献: 首先, 本文把请求页面的连续程度引入到缓存替换算法设计中, 这种基于磁盘非匀速访问特性的方法与以往的基于页面的新旧程度 (Recency) 以及访问频率 (Frequency) 的方法有着明显的不同. 其次, 多年来命中率已经成为指导和评测缓存系统性能事实上的标准, 然而命中率并不能反映失效开销对存储系统性能的影响. 本文指出这一问题并采用了综合命中率和失效开销两种因素的有效访问时间作为性能评测指标. 最后, 本文提出一种实际可用的缓存替换算法, 可以降低有效访问时间, 并通过理论分析和模拟实验验证了算法的高效性与可行性.

2 算法原理与设计

2.1 核心思想

本文提出的基于顺序检测的双队列缓存替换算法 CRASD(dual queues cache replacement algorithm based on sequentiality detection), 与其他算法相比, 具有较低的缓存失效开销, 因此能显著降低有效访问时间. 其核心思想十分简洁: CRASD 获取缓存中的页面的物理地址, 把物理地址连续的页面放入同一个集合中 (这样的集合叫做顺序集), 从而缓存中所有的页面的被划分成为若干个不相交的顺序集; 发生缓存替换时, CRASD 在含有最多元素的顺序集中选择具有最小物理地址的页面进行淘汰.

CRASD 能够有效工作, 其原因在于不同情况的缓存失效引起的失效开销是不同的: 对于在磁盘上连续分布的页面, 仅首页面的失效开销较大, 后续页面的失效请求将可以被快速满足; 而对于在磁盘上随机分布或者连续程度不大的页面, 其失效请求被满足则要慢得多, 因为后者会引发大量的磁头寻道和旋转操作, 这种磁头寻道和旋转的开销要比单纯的数据传输代价大得多. 同时, 由于应用程序的访问模式具有一定的时间稳定性, 也就是说, 以前顺序访问的页面将来往往也是被顺序访问, 以前随机访问的页面将来往往也是被随机访问的. 因此, 对于缓存中连续程度比较高的页面, 其将来被连

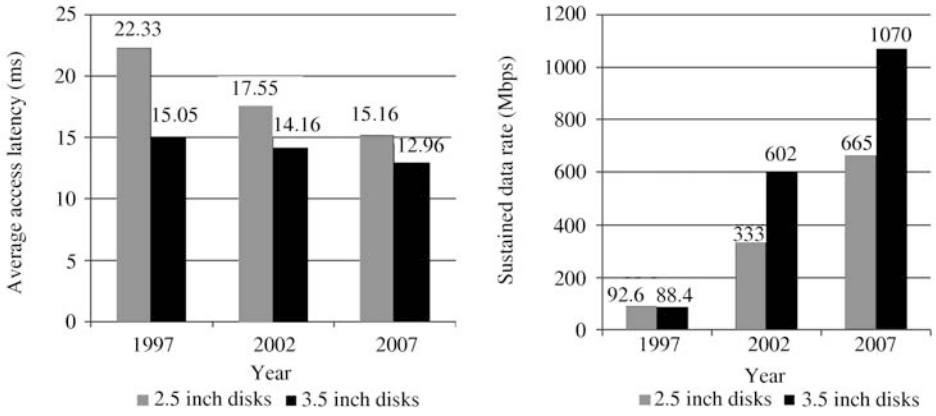

图 1 从 1997 到 2007 年磁盘性能的变化

Figure 1 Performance evolution of hard disks from the year 1997 to 2007

续访问的概率也会比较大, 由于淘汰而引发的再次访问开销就会相对比较小, 从而应该被优先淘汰.

2.2 顺序集的检测

顺序集定义为具有连续磁盘地址的一簇缓存页面的集合. 它具有两重含义: 同一顺序集的缓存页面的磁盘地址分布在一连续的地址空间; 同时, 在此特定地址空间内所有的缓存页面必定属于同一顺序集. 如果把顺序集全部页面磁盘地址中最小的地址定义为首地址 (head address), 而把顺序集中所含缓存页面的数目定义为顺序度 (sequentiality), 那么一个顺序集可以用二元属性对 ⟨head address, sequentiality⟩ 来表示. 由于文件系统通常使用线性的逻辑块地址 (logical block address, 即 LBA) 来表示和管理磁盘空间, 在文章的后面部分中使用 LBA 表示页面地址.

CRASD 把缓存中的页面划分为若干个不相交的顺序集. 当一个新页面请求发生时, 依赖于新请求页面的地址, 最多有两个顺序集会发生变化. 图 2 显示了在一个新页面请求发生的前后, 顺序集的变化情况, 图中的每一个方框代表一个缓存页面, 方框内的数字代表缓存页面的 LBA, 同一种图案的方框属于同一个顺序集.

2.3 顺序集的管理

既然每一次请求新的页面时, 顺序集都会发生变化, 那么设计一种合理的数据结构对顺序页面进行管理是影响 CRASD 算法效率的关键问题. 由于 CRASD 的页面淘汰策略是在含有最多元素的顺序集中选择页面进行替换, 我们将所有顺序集按照顺序度进行降序排列. 该队列的长度对应于缓存中的顺序集个数, 由于每个顺序集中会有大量的页面, 所以这个队列的长度相对较短, 因而对顺序集队列的维护开销是可以容忍的.

当缓存失效发生时, 如果缓存中已经没有空闲空间容纳新的页面, 通过获得这个降序队列中第一个顺序集的首地址属性, 然后淘汰位于这个首地址的页面. 这种数据结构, 避免了选择淘汰页面时引入额外的查找开销. 但是, 为了使队列一直维持在降序状态, 每次缓存失效都要更新队列, 这将引入额外的计算开销. 为了进一步降低这种开销, 算法中使用平衡二叉树来管理所有的顺序集. 图 3 显示了

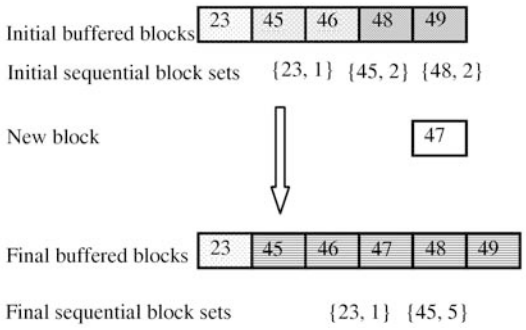

图 2　顺序集示例

Figure 2　An example of sequential block sets

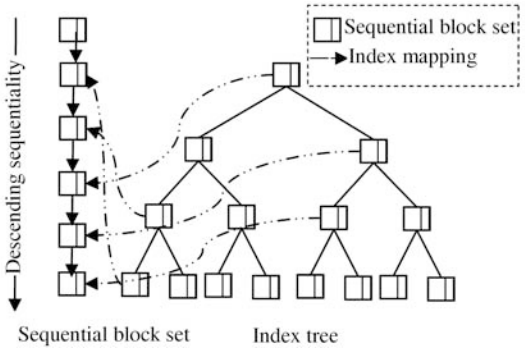

图 3　管理顺序集的数据结构

Figure 3　Data structure of management

管理顺序集所使用的数据结构. 由于平衡二叉树的查找、插入和删除开销均为 $O(\log n)$[3], 所以 CRASD 算法的时间复杂度是 $O(\log n)$ 量级, 与常用的 LFU[4], LRU-K[5] 以及 LRFU[6] 等算法相当.

2.4　算法描述

CRASD 首先把缓存页面划分为若干个不相交的顺序集, 然后从顺序度最高的顺序集中选择页面进行淘汰. CRASD 通过维护一个按顺序度降序排列的优先队列 Q 对顺序集进行选择与管理, 当缓存替换发生时, 淘汰 Q 中的第一个顺序集中具有最小地址的页面. 当某一个顺序集被选中淘汰时, 它的顺序度会不断减小, 而其他集合的顺序度可能会增大, 甚至大于正在被淘汰集合的顺序度. 然而, 为了降低将来可能引起的失效开销, 一旦选中某个顺序集进行淘汰, 最好先依次把这个集合中的页面淘汰出去, 而不总是从顺序度最大的集合中选取淘汰页面. CRASD 使用了一个指针指向了当前正在替换的顺序集 $\mathrm{PS_{rep}}$, 确保旧的集合中所有的页面被淘汰之前, 不会有新的集合被选中. 同时, 对于正在被访问页面所属的顺序集, 即使它的顺序度最大, 也不应该首先被淘汰出去. CRASD 使用了另外一个指针指向当前正在访问的顺序集 $\mathrm{PS_{req}}$, 以保护程序正在使用的页面.

当对页面 p 的访问请求在缓存中命中时, CRASD 执行一个过程来保护 p 所属的顺序集, 避免它被过早地替换掉. 如果 p 属于 Q 中的第 i 个集合 $Q[i]$, 则把 $Q[i]$ 标记为当前正在访问的顺序集 $\mathrm{PS_{req}}$. 如果 $Q[i]$ 正在被替换, 即 $Q[i]$ 等于集合 $\mathrm{PS_{rep}}$, 则同时停止替换 $Q[i]$ 中的页面, 即把 $\mathrm{PS_{rep}}$ 置为空.

当对页面 p 的访问请求在缓存中失效时, 如果缓存中存在空闲, 就分配缓存空间, 装载 p 到缓存中. 由于有新的页面装载入缓存, 首先, CRASD 需要更新顺序集. 基于页面 p 的地址, 可能会发生 3 种情况: 创建一个新的顺序集, 扩展一个已有的顺序集, 或者是把已有的两个顺序集合并为一个顺序集 (第 3 种情况如图 2 所示). 然后, CRASD 调整优先队列 Q, 维持 Q 中的顺序集按照顺序度降序排列. 最后, 同缓存命中的情况一样, CRASD 也要执行一个同样的过程来保护 p 所属的顺序集. 在对页面 p 的访问在缓存中失效, 而缓存中不存在空闲的情况下, 需要在装载页面 p 入缓存之前先执行一个页面淘汰过程: 如果 $\mathrm{PS_{rep}}$ 为空, 需要判断当前正在访问的顺序集 $\mathrm{PS_{req}}$ 是否指向 Q 中第一个集合 $Q[1]$, 如果是, 则把第二个集合 $Q[2]$ 设定为替换集合 $\mathrm{PS_{rep}}$, 否则把 $Q[1]$ 设定为替换集合 $\mathrm{PS_{rep}}$, 之后, 则可以淘汰 $\mathrm{PS_{rep}}$ 中具有最小磁盘地址的页面; 如果 $\mathrm{PS_{rep}}$ 非空, 则直接淘汰 $\mathrm{PS_{rep}}$ 中具有最小磁盘地址的页面.

对上述算法进行模拟实验, 我们发现在一个短期的运行系统中, 此算法性能良好, 而在一个长期的运行系统中, 此算法的性能则不那么令人满意. 主要原因是上述算法总是把小簇的页面保留在缓存

```
input: a new page p
initialization: a replacement queue Q1 and an
access queue Q2, an index tree T

if p is not in cache {
  if cache is full {
    if Q1 is empty {
       switch Q1 and Q2
    }
    if PS_rep is empty or is null {
       set PS_rep=((PS_req!=Q1 [1])?Q1[1]:Q1[2])
    }
    drop the page PS_rep.HA
    PS_rep.SE−−
    PS_rep.HA++
  }
  put p in the freed slot
  AdjustQueue (Q2, p)
  Adjust Q1 to maintain the descending order
  ProtectRecent (Q1, p)
} else {
       ProtectRecent (Q1, p)
}
```

```
AdjustQueue(Q, p) {
   lookup the index tree T to find such indexs i, j:
   if (Q[j].HA=p.LBA+1) & (Q[i].HA+Q[i].SE=p.LBA) {
        update Q[i].SE += Q[j].SE+1
        remove Q{j} from Q
        } else if (Q[i].HA=p.LBA+1) {
        update Q[i].HA=p.LBA
        update Q[i].SE+=1
        } else if (Q[i].HA+Q[i].SE=p.LBA) {
        update Q[i].SE+=1
        } else {
        Add a new sequential page set BS_new to Q
        set BS_new.HA=p.LBA
        set BS_new.SE=1 }
     Adjust Q to maintain the descending order
     }
```

```
ProtectRecent (Q, p) {
   if p is in PS_rep {
        set PS_req=PS_rep
        set PS_rep=null
   } else if p is in Q[1] {
        set PS_req=Q[1]
   }
```

图 4 CRASD 的伪代码. SE 代表顺序集的顺序度, HA 代表顺序集的首地址, LBA 代表页面地址

Figure 4 Pseudo-code of CRASD algorithm

中, 而系统经过长期的运行后, 大量的小簇页面占据了过多的缓存, 反而引起了性能下降. 为了解决这一问题, 我们把优先队列 Q 划分为两个队列: 替换队列 $Q1$ 和新请求队列 $Q2$. 初始缓存为空的状态下, $Q1$ 和 $Q2$ 都为空. 在缓存还没有首次填满时, 新请求页面加入到 $Q1$ 中. 当缓存满了以后, 需要空闲空间时, 从队列 $Q1$ 中淘汰页面, 新请求页面则加入队列 $Q2$ 中. 当替换队列 $Q1$ 为空的时候, $Q1$ 和 $Q2$ 进行交换. 这样可以始终从 $Q1$ 中淘汰页面, 而新页面也始终加入到 $Q2$ 中. 双队列缓存替换算法的伪代码如图 4 所示.

在 CRASD 算法中, 优先淘汰缓存中顺序度最高的页面序列. 对于只读一次的长顺序页面序列, 它们会很快从缓存中被淘汰出去, 而不是占用失效代价更大的随机页面的缓存空间. 对于那些会重复访问的长顺序序列, 它们也会很快从缓存中被淘汰出去. 幸运的是, 预取算法可以保证重新取回这些顺序页面的性能和效率. 事实上, 顺序访问也并不是存储系统的性能瓶颈. 对于随机页面来说, 它们将尽可能地被保留在缓存中, 直到它们聚集成为比较大的簇, 或者在 $Q1$ 中没有更加连续的页面存在.

3 性能分析

3.1 示例分析

本小节使用一个例子证明, 以有效访问时间为衡量标准, CRASD 有达到比 Belady's MIN[7] 更好

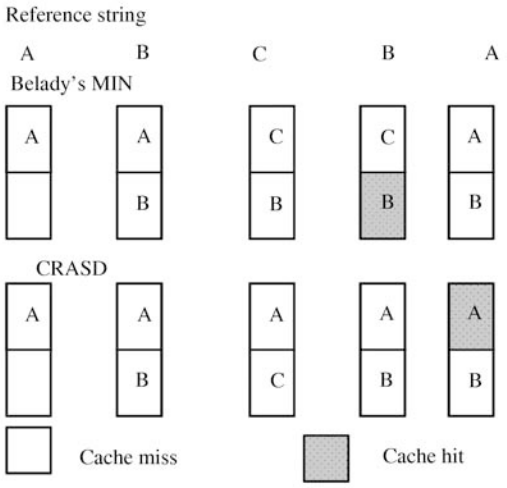

图 5　两种算法缓存内容变化对比

Figure 5　The change of page frames

表 1　性能分析所使用的符号标记

Table 1　Notations used in the analysis

T_{effect}	Effective access time
T_{hit}	Cache hit time, memory access time
T_{thrash}	Average seek time + average rotation time
T_{transfer}	Average transfer time of a block
N_{total}	Total number of blocks
N_{hit}	Number of hit
N_{thrash}	Number of seek and rotation
H	Hit ratio = $N_{\text{hit}}/N_{\text{total}}$
F	Fault ratio =$1-H$
S	Thrash ratio = $N_{\text{thrash}}/N_{\text{total}}$

性能的潜力. 下一小节会给出具体的理论分析. 假设有以下访问序列 {A, B, C, B, A}, 其中 A 包含了总大小为 20 MB 随机分布在磁盘上的页面, 而 B 和 C 分别包含了不同的 20 MB 顺序分布在磁盘上的页面, 并且 3 个集合是不相交的. 为了简化我们的分析, 假设读缓存的大小为 40 MB. 在这个读缓存中分别使用 CRASD 和 Belady's MIN 替换策略以后, 缓存中页面的变化情况如图 5 所示.

从图 5 中可以看到, Belady's MIN 和 CRASD 算法具有相同的缓存命中率. 然而, 有效访问时间却有很大的差别. 假设 A 中页面的平均大小为 4 KB, 并且在实验中使用希捷的 Barracuda 7200.11(Model Number: ST31000340AS) 磁盘做为存储介质. 通过简单的计算可以得出, CRASD 的聚合访问时间为 67.12 s, 而 Belady's MIN 的聚合访问时间为 133.47 s.

3.2　理论分析

本文使用有效访问时间来评价替换算法的性能. 有效访问时间, 反映了应用程序从发出数据请求到收到数据应答所需要的时间, 是一个比命中率更优的综合指标. 当数据请求在缓存命中时, 页面直接被送往应用程序, 这时有效访问时间等同于内存访问时间. 当在缓存中失效时, 页面需要首先从磁盘读入内存, 然后再送往应用程序. 磁盘读取时间通常分为三部分: 块传输时间、寻道时间和旋转时间. 在分析中我们使用表 1 所示符号标记.

为了简化分析, 本文不区分寻道时间和旋转时间. 每次磁头需要移动以定位数据块的时候, 我们称其为一次抖动. 我们把抖动时间定义为磁头从当前块移动到目标块所需要的时间. 既然寻道和旋转反映了磁头的移动行为, 抖动时间可以认为等于寻道时间和旋转时间的平均值之和. 基于上述定义, 有效访问时间可以表示如下:

$$\begin{aligned} T_{\text{effect}} &= (\text{总内存访问时间} + \text{总块传输时间} + \text{总抖动时间})/\text{总数据块数} \\ &= (T_{\text{hit}} * N_{\text{hit}} + (N_{\text{total}} - N_{\text{hit}}) * (T_{\text{transfer}} + T_{\text{hit}}) + N_{\text{thrash}} * T_{\text{thrash}})/N_{\text{total}} \\ &= T_{\text{hit}} + (1 - N_{\text{hit}}/N_{\text{total}}) * T_{\text{transfer}} + (N_{\text{thrash}}/N_{\text{total}}) * T_{\text{thrash}} \\ &= T_{\text{hit}} + (1 - H) * T_{\text{transfer}} + S * T_{\text{thrash}}. \end{aligned}$$

T_{hit}, T_{transfer} 和 T_{thrash} 都是取决于硬件设备的常量, 所以要降低有效访问时间需要提高命中率或者降低抖动率. 然而, 改进这两种因素的难易程度是不同的. 假设有两种不同的缓存替换算法 A 和 B, 二者都把有效访问时间从 $T_{\text{effect}-\text{old}}$ 改进到 $T_{\text{effect}-\text{new}}$. 对于算法 A, 命中率从 H_{old} 提高到 H_{new}, 而抖动率 S_{old} 保持不变. 对于算法 B, 抖动率从 S_{old} 降低到 S_{new}, 而命中率 H_{old} 保持不变. 可以得出:

$$(S_{\text{old}} - S_{\text{new}})/(H_{\text{new}} - H_{\text{old}}) = T_{\text{transfer}}/T_{\text{thrash}}.$$

以磁盘 Barracuda 7200.11 为例, 其持续传输速率为 105 MB/s, 平均访问延迟约为 13 ms. 由于平均的块传输时间反比于持续传输速率, 可以算出 T_{thrash} 是 T_{transfer} 的数千倍, 从而降低抖动率要比提高命中率对性能的改善更有效. 随着磁盘工业的不断发展, T_{thrash} 与 T_{transfer} 之间的差距会越来越大 (如图 1 所示), 降低抖动率也就显得更加重要.

更进一步, 假设使用算法 A 后, 命中率提高了 x 个百分点, 抖动率不变; 使用算法 B 后, 抖动率降低了 y 个百分点, 命中率不变; 二者仍然达到相同的有效访问时间, 则有

$$y/x = (T_{\text{transfer}}/T_{\text{thrash}}) * (H_{\text{old}}/S_{\text{old}}).$$

$T_{\text{transfer}}/T_{\text{thrash}}$ 是取决于磁盘性能的一个常数, 仍以 Barracuda 7200.11 为例, 这个值大约是 1/2791. 这意味着当 $H_{\text{old}}/S_{\text{old}}$ 等于 2791 时, 对抖动率改进的每一个百分比等价于对命中率改进的每一个百分比. 而如果 $H_{\text{old}}/S_{\text{old}}$ 只有 279, 则对抖动率改进的每一个百分比等价于对命中率改进了 10 个百分比.

既然只有当缓存失效的时候才会发生抖动, 那么增大缓存对降低抖动率无疑是有效的. 然而出于成本预算的原因, 增大缓存容量并不容易. 另一个降低抖动率的方法就是避免非顺序块访问时发生的缓存失效. 对一个长的顺序页面序列访问, 缓存失效时只会产生一次抖动, 而对于许多非顺序页面访问, 缓存失效时会产生多次抖动. CRASD 优先淘汰顺序页面, 保留非顺序页面, 所以它可以显著降低抖动率. 另一方面, 由于在文件系统和存储服务器中广泛应用预取算法, CRASD 的命中率下降比较小. 所以, CRASD 可以降低有效访问时间, 从而提高整个缓存系统性能.

4 实验结论

本文所使用的 trace 收集于一台运行了 ftp 文件服务、apache tomcat web 服务、mysql 数据库以及其他工具的活跃的 Windows NT 网站服务器中. 这些 trace 收集于几个月的时间段内, 它们曾经用在以往研究工作 [8,9] 的性能评测中. 其中, ftp 文件服务的访问模式主要是顺序访问, apache tomcat web 服务的访问符合 zipf 分布, mysql 数据库服务的访问模式介于顺序访问和随机访问之间, 而整个网站服务器的访问则具有由多用户发起的多任务混合访问模式, 具有一定的代表性和普遍性. 我们选择了 10 个典型的 trace 代表不同的工作负载. 选中的 trace 的特性如表 2 所示, 页面大小为 512 字节.

在模拟实验中, 我们实现了 CRASD、LRU[10] 和 ARC[11] 三种算法. LRU 是应用最广泛的替换算法, 在长期的负载中具有优秀的性能; 而 ARC 是近年来公认的最优的单机缓存替换算法, 已经用于 IBM DS 8000 系列存储服务器中. 我们使用 CMU 开发的 DiskSim 对硬盘进行模拟, 利用希捷 Barracuda7200.11 的规范给硬盘建模, 其平均寻道时间为 8.5 ms, 平均旋转时间为 4.16 ms. 我们把缓存从 8 MB 逐渐增加到 40 MB, 然后记录下对应的有效访问时间, 其结果如图 6 和 7 所示.

从图 6 和 7 可以看出, CRASD 的有效访问时间比 LRU 和 ARC 降低了 5%–15%, 这是符合理论预期的. 此外, 我们还发现了一种异常现象. 在 T4 的实验结果中, 当缓存容量从 16 MB 提高到 24 MB

表 2　实验中所使用的 trace 特性

Table 2　Characteristics of traces used in our simulations

Trace name	Number of requests	Unique blocks
T1	17 483 577	6 130 732
T2	14 002 305	4 379 329
T3	14 564 573	7 194 783
T4	1 125 822	665 652
T5	901 457	549 212
T6	1 452 589	555 543
T7	1 554 122	888 576
T8	1 325 576	733 191
T9	1 145 897	622 772
T10	1 247 781	591 202

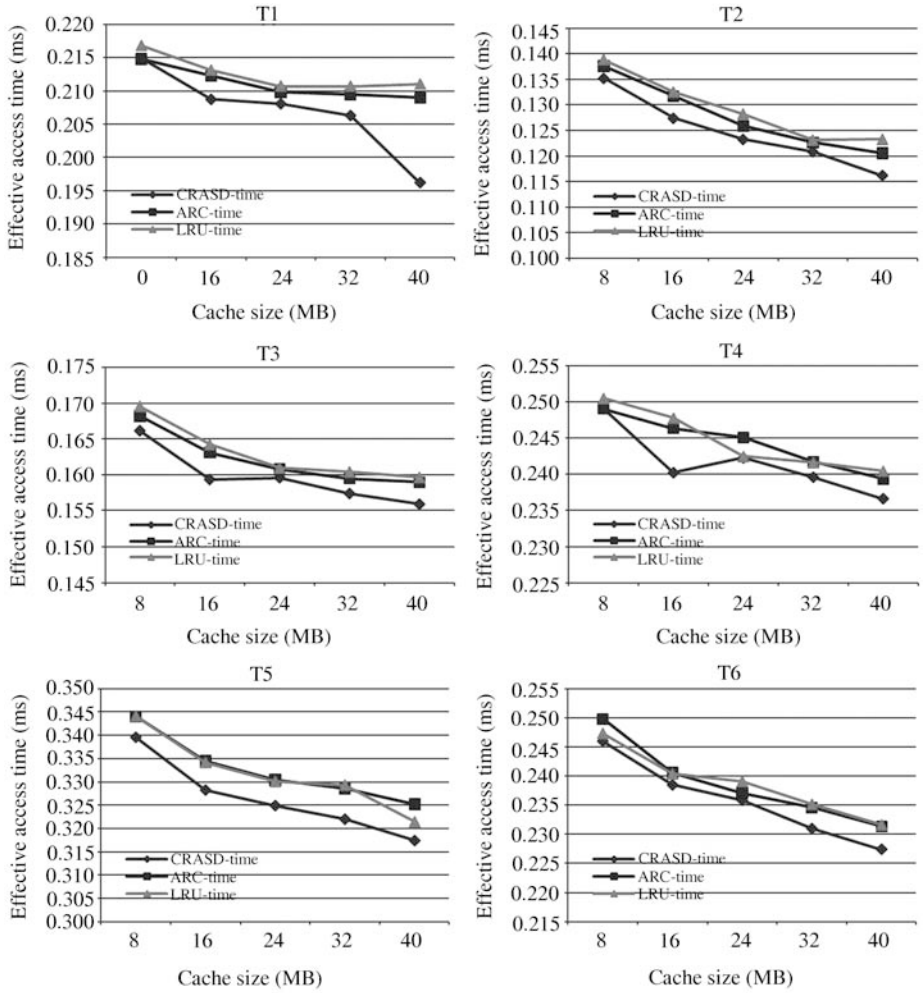

图 6　CRASD、ARC 和 LRU 三种替换算法的有效访问时间比较 (a)

Figure 6　Effective access time per block achieved by CRASD, ARC and LRU(a)

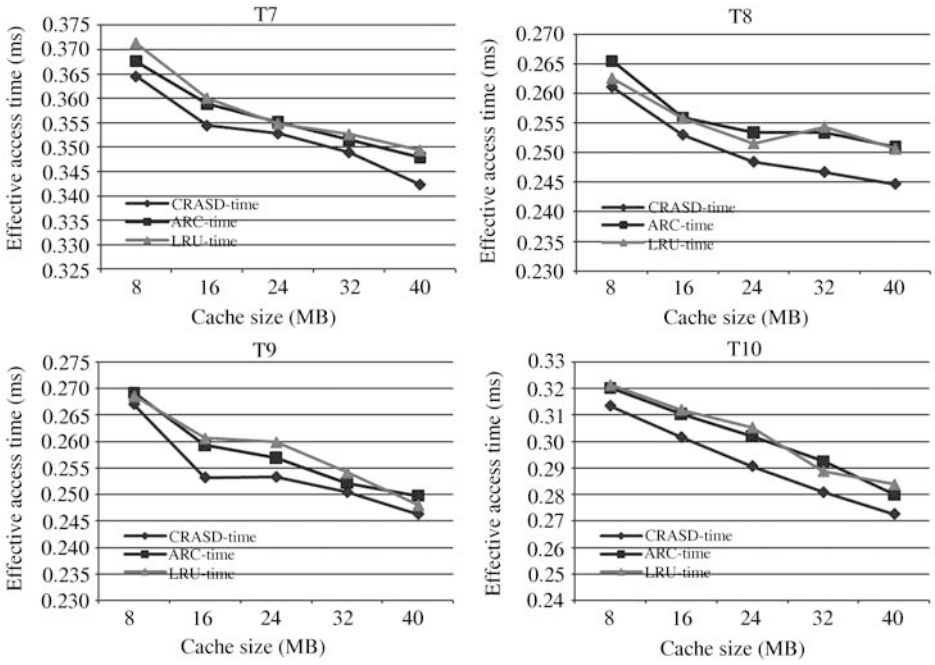

图 7 CRASD、ARC 和 LRU 三种替换算法的有效访问时间比较 (b)

Figure 7 Effective access time per block achieved by CRASD, ARC and LRU(b)

时, CRASD 的有效访问时间非但没有降低, 反而升高了. 在 T8 的实验结果中, 当缓存容量从 24 MB 提高到 32 MB 时, LRU 的有效访问时间也升高了. 我们可以推断出, 在某些情况下, ARC 也会发生类似的情况. 由此可见, 提高缓存的容量, 并不总能降低有效访问时间. 这是因为访问缓存可能使得原本的只需要一次寻道的数据访问需要多次寻道. 例如对于访问序列 {200–300, 1–1000}, 没有缓存时, 只需要 2 次寻道; 如果有缓存, 恰好能容纳页面 {1–300}时, 反而可能产生 3 次寻道. 设计一种缓存替换算法避免这种现象是一种新的挑战.

5 相关工作

缓存替换算法已经得到了深入广泛的研究. 早期的替换算法集中于开发局部性, 以提高缓存的命中率. Belady 提出一种离线算法 MIN, 给出了命中率的理论上限. LRU 算法和 LFU 算法是在线算法的典型代表, LRU 淘汰最长时间未使用过的页面, LFU 淘汰使用频率最低的页面.

后期的算法大多建立在 LRU 和 LFU 的基础之上. LRU-K 利用倒数第 K 次访问的逻辑时间到现在的时间距离选择淘汰页面. 2Q[12] 维护两条队列分别管理访问一次以及超过一次的页面, 根据阀门值选择队列淘汰页面. MQ[13] 使用多条 LRU 队列管理不同访问频率的页面, 它在有最小队列序号的非空队列中淘汰页面. LIRS[14] 统计最后两次连续访问之间访问的不同页面数, 据此把页面分为 LIR 和 HIR 两个集合, 从 HIR 中淘汰最老的页面. ARC 使用两条 LRU 队列区分一次访问与多次访问的页面, 并采用两条附加队列保存已淘汰页面的历史信息, 可根据历史信息的变化自动适应负载的变化.

随着网络存储的发展, 诞生了针对网络环境下多级缓存结构的替换算法. ULC[15] 算法对所有级别的缓存实行集中控制, 它通过发出 Retrieve 和 Demote 指令控制页面在不同层次的缓存之间迁移. 针对某些特殊应用, Karma[16] 利用应用程序提供的信息, 控制页面在缓存中的分布. DEMOTE[17] 提

出在高层缓存淘汰以前, 将页面发送到低级缓存中. PROMOTE[18] 利用概率过滤技术, 把页面从低级缓存提升到高级缓存中.

上述研究的工作原理是时空局部性, 以提高命中率为目标. 针对磁盘访问进行优化, 能降低寻道和旋转时间的工作是 STEP[19], SARC[20] 以及 Bargain Cache[8]. STEP 和 SARC 偏重于预取, 可以与 CRASD 互为补充, 而 Bargain Cache 则是文件级的缓存替换策略, 缺少了 CRASD 块级策略的灵活性与通用性.

6 结束语

传统的缓存替换算法研究往往忽略了磁盘访问特性. 磁盘与内存不同, 在传输数据之前, 磁头需要进行耗时的寻道和旋转操作. 访问连续页面只需要一次寻道与旋转操作, 而访问同样大小的多个非连续页面, 则需要多次寻道与旋转操作. 因而, 顺序页面的缓存失效开销要比多个非顺序页面的失效开销小. 基于上述原理, 本文将页面的顺序程度引入到缓存替换算法设计中, 提出了基于顺序检测的双队列缓存替换算法, 从降低缓存失效开销入手来提高缓存系统整体性能. 本文采用了综合命中率和失效开销两种因素的有效访问时间作为性能评测指标, 通过理论分析和模拟实验验证了本算法的高效性和可行性.

参考文献

1 Hu Y, Nightingale T, Yang Q. Rapid-cache-a reliable and inexpensive write cache for high performance storage systems. IEEE Trans Parall Distrib Syst, 2002, 13: 290–307

2 Muntz D, Honeyman P. Multi-level caching in distributed file systems-or-your cache ain't nuthin' but trash. In: Proceedings of the Winter 1992 USENIX Conference. San Francisco, 1992. 305–313

3 Adelsonveslkii G M, Landisand Y M. An algorithm for the organization of information. Doklady Akademi Nauk, 1962, 16: 263–266

4 Aho A V, Denning P J, Ullman J D. Principles of optimal page replacement. J ACM, 1971, 18: 80–93

5 O'Neil E J, O'Neil P E, Weikum G. The LRU-K page replacement algorithm for database disk buffering. In: Proceedings of the 1993 ACM SIGMOD International Conference. Washington, 1993. 297–306

6 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. IEEE Trans Comput, 2001, 50: 1352–1360

7 Belady L. A study of replacement algorithms for a virtual-storage computer. IBM Syst J, 1966, 5: 78–101

8 Zhao Y J, Xiao N. Bargain cache: using file-system metadata to reduce the cache miss penalty. In: Proceedings of the 9th PDCAT Conference. Dunedin, 2008. 177–184

9 Chu R, Xiao N, Zhuang Y Z, et al. A distributed paging RAM grid system for wide-area memory sharing. In: Proceedings of the 20th IPDPS Conference. Rhodes Island, 2006. 10–17

10 Mattson R L, Gecsei J, Slutz D R, et al. Evaluation techniques for storage hierarchies. IBM Syst J, 1970, 9: 78–117

11 Megiddo N, Modha D S. ARC: a self-tuning, low overhead replacement cache. In: Proceedings of the 2nd FAST Conference. San Francisco, 2003. 115–130

12 Johnson T, Shasha D. 2Q: a low overhead high performance buffer management replacement algorithm. In: Proceedings of the 20th VLDB Conference. Santiago de Chile, 1994. 297–306

13 Zhou Y Y, Philbin J F. The multi-queue replacement algorithm for second level buffer caches. In: Proceedings of USENIX Annual Technology Conference. Boston, 2001. 91–104

14 Jiang S, Zhang X D. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. In: Proceedings of the ACM SIGMETRICS Conference. Marina Del Rey, 2002. 31–42

15 Jiang S, Zhang X D. ULC: a file block placement and replacement protocol to effectively exploit hierarchical locality in multi-level buffer caches. In: Proceedings of the 24th ICDCS Conference. Tokyo, 2004. 168–177
16 Factor M, Schuster A, Yadgar G. Karma: know-it-all replacement for a multilevel cache. In: Proceedings of the 5th FAST Conference. San Jose, 2007. 169–184
17 Wong T M, Wilkes J. My cache or yours? making storage more exclusive. In: Proceedings of the USENIX Annual Technical Conference. Monterey, 2002. 161–175
18 Gill B S. On multi-level exclusive caching: offline optimality and why promotions are better than demotions. In: Proceedings of the 6th FAST Conference. San Jose, 2008. 49–65
19 Liang S, Jiang S, Zhang X D. STEP: sequentiality and thrashing detection based prefetching to improve performance of networked storage servers. In: Proceedings of the 27th ICDCS Conference. Toronto, 2007. 550–559
20 Gill B S, Modha D S. SARC: sequential prefetching in adaptive replacement cache. In: Proceedings of the USENIX Annual Technical Conference. Anaheim, 2005. 293–308

Dual queues cache replacement algorithm based on sequentiality detection

XIAO Nong*, ZHAO YingJie*, LIU Fang & CHEN ZhiGuang

Department of Computer Science, National University of Defense Technology, Changsha 410073*, China*
*E-mail: nongxiao@nudt.edu.cn, hyperseymour@163.com

Abstract Caching is one of the most effective and commonly used mechanisms to improve performance of storage servers. Replacement policies play a critical role in the cache design due to the limited cache capacity. Recent researchers devote themselves to achieve high hit ratios, but rarely pay attention to reducing miss penalty during the design of a replacement policy. To address the issue, this paper presents a novel algorithm, called dual queues cache replacement algorithm based on sequentiality detection, which prefers to drop sequential blocks and protect random blocks. The buffer cache can serve more subsequent random read requests, so the cache miss penalty decreases significantly. Moreover, the algorithm makes use of two queues separately maintaining new blocks and old blocks to avoid the degradation of hit ratios. Our trace-driven simulation results show that it performs better than LRU and ARC for a wide range of cache sizes and workloads.

Keywords caching mechanism, replacement policy, hit ratio, miss penalty, sequentiality detection