一种新的基于 RAID 的 CACHE 技术研究与实现

易法令[1,2]，谢长生[1]，万继光[1]

[1](华中科技大学 计算机学院 国家外存储重点实验室，湖北 武汉 430074)

[2](荆州师范学院 计算机系，湖北 荆州 434000)

摘 要：设计了一个新的基于 RAID 的 Cache 系统，本系统有如下创新之处：(1)在全局配置 Cache 时，采取了预留与动态分配相结合的策略，并建立了分配模型，从理论上确定了预留与动态分配的最佳比例 (2)建立了二级读 Cache 结构，其中：一级读 Cache 实现时间的局部性；二级读 Cache 实现空间的局部性 (3)提出了定时搬移并按阈值淘汰的策略，即定时把二级 Cache 的节点中最近访问过的数据小块搬移到一级 Cache，当搬移的数据小块超过一个阈值时，淘汰二级 Cache 的相应节点 通过仿真测试证明了定时搬移策略较好地实现了时间的局部性，提高了 Cache 系统的性能

关 键 词：Cache；RAID；定时搬移；淘汰搬移；LUN

中图分类号：TP311 **文献标识码**：A **文 章 编 号**：1000-1220(2004)12-2173-06

Research of A New Cache Technology on RAID

YI Fa-ling[1,2], XIE Chang-sheng[1], WAN Ji-guang[1]

[1](*National Storage System Laboratory, Huazhong University of Science and Technology, Wuhan* 430074, *China*)

[2](*Department of Computer of Jingzhou Normall School, Jingzhou* 434000, *China*)

Abstract: Designsed a new cache system on RAID. This system has the following innovations: (1)Adopting private and dynamic allocation strategy in Deploying cache, and deciding the prime proportion of the private and the public through constructing allocating model (2)Constructing two-level read cache structure In this structure, level 1 read cache exploits temporal locality, and level 2 read cache exploits spatial locality. (3)Presenting timing-move and replacing according to a given threshold strategy. This strategy is moving the small data blocks which are accessed lately in level 2 cache's node to level 1 cache, and replacing the level 2 cache's node when small data blocks moved exceeds a given threshold. The paper still proves that timing-move strategy exploits temporal locality effectively and improve the cache system perfomance through simulation test.

Key words: Cache; RAID; timing-move; replacing-move; LUN

1 引 言

随着存储时代的到来，廉价冗余磁盘阵列(RAID)技术日益凸现其重要性 其大容量的存储和较高的可靠性以及低廉的价格等诸多优势，使其成为大多数存储方案的首选 但是，RAID 的 IO 性能的不足，在一定程度上制约了 RAID 的规模和性能，因此，采用 Cache 技术提高 RAID 的 IO 性能一直是 RAID 研究的重要方向[1-4]。

在实践中，对 Cache 的设计和研究主要集中在以下几个方面：

(1) Cache 的全局配置问题；

(2) 读 Cache 的层次结构问题；

(3) 读、写 Cache 的关系及写 Cache 的可靠性；

(4) 写 Cache 的 Destage[3](把写 Cache 中的"脏"块淘汰到磁盘)等

文献[5]提出了一种全局按逻辑单元(LUN)固定配置 Cache，即每一个 LUN 固定使用一块内存区 如此分配，避免了不同 LUN 用户使用 Cache 时，某个用户抢占过多的 Cache 资源而影响其它用户的性能[5]。但是，当不同的 LUN 用户使用不均时，很可能造成大量 Cache 资源闲置 文献[6]介绍了一个 RAID 的两级读 Cache 模式，通过小块的一级 Cache 实现时间的局部性，通过大块的二级 Cache 实现空间的局部性 在二级 Cache 淘汰时，把被访问过的小数据块有选择性的搬迁到一级 Cache[6]。该算法虽然在一定程度上满足了 Cache 的时间/空间局部性的要求，但是在具体实现方面还是有一定的缺陷 因为在一般的设计中二级 Cache 要比一级 Cache 大得多，如果二级 Cache 很大，并采用一般的 LRU 或 LFU 淘汰算法，那么很可能被搬迁到一级 Cache 中被访问的块是较长时间以前被访问的块，虽然该方法延长了被访问块的生命期，但是并不符合时间局部性的要求，这一点在后面的实验评估中将被验证 在写 Cache 的 Destage 算法方面，比较典型的是高/低水位算法[4]和线性阈值 Destage 算法[7]。前者

是设置淘汰的上、下限, 即超过上限开始淘汰, 直到下限停止淘汰; 采用这种算法, 上、下限往往不好确定: 太高, 当短时间内有大量写请求时会很快写满 Cache, 导致拥塞; 太低, 写 Cache 又不能得到充分利用. 后者是根据 Cache 的占用状态确定 Destage 的速度, 即: Cache 的占用率高时, Destage 速度快; 反之, 则慢. 该算法虽然在一定程度上解决了第一种方法的问题, 但是当写 Cache 占用比较大时, 往往是写请求比较密集的时候, 提高后台 Destage 的速率, 势必会降低前台对外来请求的响应速度, 从而影响整体的 I/O 性能.

本文通过分析 RAID 的存储特性和借鉴国内外在磁盘 Cache 以及处理器 Cache 方面的有关理论和经验[8,9]研究并设计了一个效率较高的 Cache 系统. 与以往的 RAID 控制器的 Cache 相比, 本系统主要有如下创新或改进之处:

(1) 针对 RAID 中不同的 LUN, 采用按比例预留与全局动态分配 Cache 的方法, 既避免了因某个用户抢占了过多的 Cache 存储块, 而导致另一个 LUN 用户请求实现困难; 又减少了因不同 LUN 用户对 RAID 使用不均时, 而导致 Cache 存储资源的浪费. 同时, 建立了一个按 LUN 分配 Cache 的模型, 从理论上确定了预留和动态分配的最佳比例.

(2) 采用读写 Cache 分开的策略, 读、写 Cache 的设计上都以缩短请求响应时间为目标. 写 Cache 主要在 Destage 策略上体现这一目标; 读 Cache 则是以充分体现 Cache 的两大特性为基础. 根据 Cache 的时间局部性和空间局部性原理, 在设计读 Cache 时采用两级结构: 一级实现时间的局部性, 二级实现空间局部性.

(3) 在写 Cache 的 Destage 策略的设计中, 根据 CPU 的工作情形采用自适应的高、低水位策略. 在不影响系统对外请求的处理能力的情况下, 尽量使写 Cache 为"空", 使大量的写请求能及时响应. 限于篇幅, 本文主要详细分析前两个方面.

2 Cache 系统的结构和数据流介绍

在整个 RAID 系统的设计中, 把 Cache 与 RAID 控制部分分为两个模块; Cache 系统采用读、写 Cache 分开的体系结构, 其中, 读 Cache 又分为一级读 Cache、二级读 Cache, 其系统结构和数据流图如下(图 1). 如此设计有如下优势:

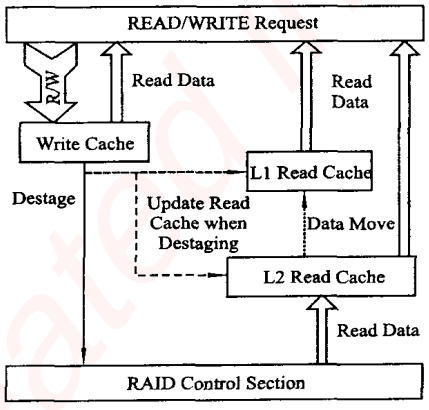


图 1 Cache 系统结构与数据流图

Fig. 1 Architecture and data-flow of cache system

(1) 提高系统的写性能: 与系统写相关的一个关键问题就是奇偶校验信息如何处理. 由于奇偶校验信息只是在写数据到磁盘时使用, 因此, 它只是与写 Cache 相关. 在组织写 Cache 数据时以分条(Stripe)为单位, 可以把一个 Stripe 中的小块数据合并为一个满 Stripe, 即"小写变大写". 这样在 Destage 时尽量淘汰满 Stripe, 克服小块写设计中奇偶校验信息数据的改写, 只是在特殊情况下, 没有满 Stripe 或使用透写方式时, 才从磁盘中读取奇偶校验信息数据.

(2) 降低成本: RAID 的可靠性是设计中要注意的一个关键问题. 为保证可靠性, 常规方法是采用软/硬件冗余的方式. 由于读数据出错不会导致数据丢失, 为了节约 CACHE, 通常只是对"写 CACHE"的内容进行双备份, 这样能降低系统的成本.

(3) 在读 Cache 的两级结构中, 能有效地实现时间与空间的局部性.

从图 1 中可以看出系统处理数据的流程是: 当有写请求时, 直接把数据写入写 Cache, 在写 Cache 的 Destage 线程工作时, 把写 Cache 中的"脏"块写回到 RAID, 同时又更新一级读 Cache、二级读 Cache 中相同地址的内容. 当有读请求时, 首先查找写 Cache, 然后查找一级读 Cache, 最后查找二级读 Cache, 直到把数据读入.

3 Cache 配置策略

3.1 Cache 的整体配置

RAID 系统一般按 LUN 配置, 针对不同的 LUN 有不同的用户或用户群, 在设计 Cache 时要既要考虑三种 Cache 的不同特点, 又要考虑不同 LUN 之间的均衡性. 写 Cache 采用全相联映射方式, 即写入磁盘中的数据可以对应到 Cache 中的任何位置, 没有映射规则的限制. 由于写 Cache 有一个 Destage 后台线程, 一般情况下总是有空闲的写 Cache 块, 因此, 不需考虑不同 LUN 使用 Cache 不均衡的问题. 一级读 Cache 采用组相联映射方式, 数据块的替换和查找采用 Hash 算法, 在设计 Hash 算法时要同时兼顾了查找效率和 LUN 使用 Cache 的均衡性. 二级读 Cache 占用了整个 Cache 资源的大部分, 因此, 二级读 Cache 的配置至关重要.

3.2 二级读 Cache 分配算法

二级读 Cache 的分配采用了一种预留与动态分配相结合的方式: 即按比例给每个 LUN 固定预留一部分内存, 同时余下一部分内存供全局动态分配. 二级 Cache 是采用大块数据进行组织, 整个内存的分配以数据块为单位, 其查询是采用查询树的方式. 现假设二级读 Cache 一共有 N 块数据, RAID 划分为 n 个 LUN, 那么把 Cache 分成两部分, 如图 2 所示: 其中

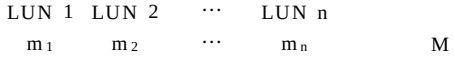


图 2 二级读 Cache 数据块分配图

Fig. 2 Allocating graph of data block for L2 read cache

$m_1, m_2, \ldots, m_n$ 分别表示与 LUN 1, LUN 2, …, LUN n 对应的预留的内存块, M 是全局动态分配的内存块. 图 2 中 m_i 只是表示内存的块数, 其内存地址并不固定. 也就是说, 对每一个 LUN 只需用一个计数器来记录其分配数据块的多少, 而不管其在什么地方. 假如 LUN i 有读请求, 需要将磁盘中对应的数据读到 Cache, 其内存具体分配方法可描述如下(设 C_i、C_M 分别表示与 LUN i 及动态分配部分对应的计数器):

```
if C_i < m_i
    在其预留的内存部分分配数据块;
else if C_M < M
    在动态分配部分分配数据块;
else
    采用全局淘汰的方法淘汰一个数据块
```

其中, 淘汰算法一般采用 LRU 或 LFU 算法, 在全局维护 LRU/LFU 链表. 由于采用了预留机制, 所以不能完全按 LRU/LFU 算法所选择的节点进行淘汰, 其淘汰算法描述如下:

```
根据 LRU/LFU 算法选择要淘汰的节点 N_i ∈ LUN i;
while C_i <= m_i
{
    选择下一个要淘汰的节点 N_j;
    i = j;
}
```

算法中, i 表示 LUN 的编号, 如果选择淘汰的节点是属于某一个 LUN 的, 则要判断该 LUN 是否用完了其预留的 Cache 块(用计数器表示); 如果未用完, 则不能淘汰, 要选择下一个要淘汰的节点.

3.3 二级读 Cache 预留与动态分配空间的比例确定

二级读 Cache 的分配策略既保证了二级 Cache 空间的充分利用, 又不至于出现在不同时段对 LUN 的访问不均衡而造成对不同 LUN 用户的响应不均的现象. 但是如何确定预留与动态分配的比例以达到最佳效果是需要重点分析的问题. 为此, 需要建立有关模型, 进行相关的分析.

3.3.1 按 LUN 分配 Cache 模型

一个 RAID 系统可以分为多个 LUN, 对不同的 LUN 用户访问的流量一般来说是不平衡的, 即使是对同一个 LUN 在不同的时段, 其流量差别也很大. 为确定分配比例, 把整个 Cache 系统抽象成两个部分, 即高速率访问部分和低速率访问部分. 具体方法如下:

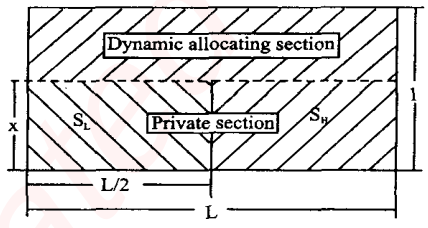


图 3 Cache 分配模型图

Fig. 3 Model graph of cache allocated

设系统有 n 个 LUN, 其容量用 L_1、L_2、…、L_i、…($1 <= i <= n$)表示, RAID 的总容量用 C 表示. LUN i 的最高访问速率与最低访问速率分别用 MAX_V_i、MIN_V_i 表示. 其中, 访问速率通过选择合适的时间间隔测试得到, 间隔不宜太小, 否则会使最高与最低访问速率差别过大, 有时甚至会使 MIN_V_i 接近 0.

根据访问的高、低速率把 Cache 分成两部分, 如图 3 所示, 其中右边表示高速率访问部分, 左边表示低速率访问部分. 高、低速率分别为 V_H、V_L:

$$V_H = \sum_{i=1}^{n} (L_i/C) * MAX\_V_i$$

$$V_L = \sum_{i=1}^{n} (L_i/C) * MIN\_V_i$$

3.3.2 Cache 读请求模型及最佳预留比例的确定

根据 Hennessy 等人的研究, 在一定的区域内 Cache 请求的命中率与 Cache 的大小大致成一种线性关系[9], 由于 RAID 的容量大大超过 Cache 的容量, 因此, 在 3.3.1 所建的模型中可以认为 Cache 工作在线性区.

设请求所占用 Cache 的容量为 S, 请求的速率为 V, 命中率与 Cache 容量的比例因子为 r, 访问时间为 t. 如果初始时 S 为 0, 那么则有如下关系式:

$$dS = V * dt - S * r * V * dt \tag{1}$$

其中 dS 表示在时间 dt 内所占用 Cache 的增加部分. 通过对(1)式进行积分变换可得到 S 与时间 t 的关系式

$$S = \frac{1 - \exp(-r * V * t)}{r} \tag{2}$$

$$t = \frac{-\ln(1 - r * S)}{V * r} \tag{3}$$

如图 3 建立的 Cache 模型, 整个 Cache 的容量为 L, $x * L$ 表示预留部分, S_H 表示高速率访问部分占用的 Cache, S_L 表示低速率访问部分占用的 Cache. 当请求以 V_H、V_L 速率访问两部分 Cache 时, 如果经过时间 t, V_H 占满了其预留部分和动态分配部分, 而此时, V_L 也恰好占满其预留部分, 则可认为 x 为最佳预留. 其原因如下: (1)如果预留小于 x, 那么 V_L 也将占用一部分动态分配部分, 这说明预留值适当增加将不会影响 Cache 的利用率; 而预留值的增加显然有利于平衡不同 LUN 用户的访问不均. (2)如果预留大于 x, 那么当 V_H 开始淘汰时 V_L 还未占满, 因此会减小 Cache 的利用率. 根据以上分析, 通过式(3)可得, 最佳预留应满足以下关系式:

$$\frac{\ln(1 - r * S_H)}{V_H * r} = \frac{\ln(1 - r * S_L)}{V_L * r} \tag{4}$$

由图 3 可得:

$$S_H = L - (L * x)/2 \tag{5}$$

$$S_L = (L * x)/2 \tag{6}$$

把(5)、(6)代入(4)得:

$$\ln(1 - r * L + r * (L * x)/2) = (V_H/V_L) \ln(1 - r * r * (L * x)/2)$$

$$L + r * (L * x)/2 = (1 - r * (L * x)/2)^{(V_H/V_L)} \tag{7}$$

在(7)中L表示整个Cache的容量; r*L表示占满全部Cache的命中率,根据按LUN分配Cache模型, r*L可以用以下方法测试得到:把全部Cache分给一半的LUN(容量占一半),然后测试这些LUN的请求在Cache完全充满时的命中率.显然r*L < 1; V_H/V_L表示最高访问速率与最低访问速率的比值,该值越大,x则越小.如果用h表示r*L,m表示V_H/V_L可得:

$$1- h+ h * x/2= (1- h * x/2)^m;$$

在实践中,只要测试出h与m的值,便可求得最佳预留x的值.

4 读Cache结构及算法分析

Cache设计的主要理论基础是时间局部性和空间局部性,为体现这两个特性,在设计读Cache时采用两级结构,即用一级Cache实现时间局部性、二级Cache实现空间局部性.因为时间局部性具有暂时性,空间局部性需要体现其空间上的扩展(预取),所以,一级Cache一般较小,其数据单位也较小,在设计中采用磁盘存取的最小单位即1个扇区(512字节);二级Cache一般较大,其对应的数据单位也较一级Cache大,在设计中采用16K字节,这样一次从磁盘中取数据至少为16K或者16K的整数倍,体现了空间局部性的要求.

在具体分析Cache的设计和算法之前,先说明请求的格式.请求最少应包含以下4个字段:(1)请求的类型,一般为读、写两种情况;(2)请求所在LUN的编号;(3)请求在LUN中的位置,用逻辑块地址(LBA)表示;(4)请求的长度.

4.1 一级读Cache算法

在处理读请求时,先查询写Cache,然后是一级读Cache,最后是二级读Cache;而一级读Cache设计是为了体现Cache的时间局部性.在设计一级读Cache时要考虑三个方面:

(1) 一级Cache的内容要充分体现及时性;

(2) 查询的效率要高于二级Cache,如果条件具备,可以使用速度更快的内存来充当一级Cache.

(3) 如何减少两级Cache中的冗余数据.

文献[6]介绍的两级Cache的结构是采用淘汰搬移的策略,即在淘汰二级Cache块时,搬移访问过的小块到一级Cache.这种策略除了不能很好地体现时间的局部性外,更重要一点是:淘汰操作一般是在处理读请求但又没有空闲块时进行,因此,"处理请求"、"淘汰"实际上是顺序操作,如果再加上"搬移"操作,显然增加了处理请求的时间.这一点在后面的实验评估中将被证明.

为体现Cache内容的时间性,在进行读请求处理时,总是定时选择二级Cache中最近访问的数据大块(节点),把大块中被访问的小块搬移到一级Cache中,为防止重复搬移,搬移后将访问标志置为未访问状态.这样,最近访问过的数据一般能在一级Cache中命中.采用这种策略,最大的问题就是数据的冗余,即在两级Cache中都存有最近访问的数据小块.为解决这个问题,采取了根据阈值淘汰的策略,即二级Cache的节点中搬移的数据小块超过一定的比例,就淘汰该节点.因为,定时搬移是通过一个单独的线程来实现的,它与"处理请求"操作实际上是一种并行操作,因此,能大大节省处理时间,同时搬移后的淘汰也减少了处理请求中的"淘汰"操作.

为提高查询效率,在进行结构设计时,一级Cache采用组相联映射方式,映射关系可以具体公式进行表达:

将CACHE存储块分成a组,每组b块,下级存储(磁盘)中以a个块为一个区来划分,设k为下级存储块存放到CACHE中的组号,j为下级存储块的块号,用LBA表示,则有如下关系:

$$gb = a \times b \qquad gb \text{ 是CACHE中的总的块数}$$
$$k = j \bmod a \qquad \cdots\cdots \tag{8}$$

通过式(8)可以确定RAID中存储块所对应的组.组内数据块n的确定则通过Hash函数来实现.Hash函数由LUN和LBA值共同决定,由于所设计的RAID最多支持256个LUN,用一个字节表示(0-255);LBA是4个字节(0-4G).所以,在实践中采用以下算法:

$$n = (LUN << 24 + LBA) \bmod b \cdots\cdots \tag{9}$$

当有数据块需要到一级Cache时,先通过(8)式计算其对应的组,然后通过(9)式计算组内的块,最后把数据插入.同样,需要查询一级Cache时,也是通过上述方法映射到对应的块,然后看是否有需要查询的数据.从这个过程很容易得出淘汰和查询算法的复杂度均为0(1).

4.2 二级读Cache设计

二级读Cache的设计要从Cache的空间局部性入手,同时也要考虑查询效率.在总体设计中,二级Cache有以下特点:

(1) 二级Cache很大,它是以大数据块为基本单位(二级Cache的数据单位是16K).

(2) 二级Cache采用预留和全局动态分配相结合的策略,并且其预留的只是Cache块的数量而未限定具体的内存地址.

基于这两点,为提高不同条件下二级Cache的利用率,我们采用全相联映射方式.

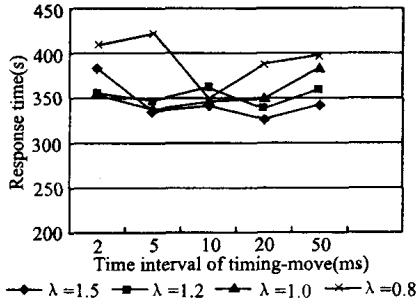
与二级Cache的查询效率直接相关的是数据块的组织方式,由于平衡二叉树具有较高的查询效率,因此,设计时是以LUN为基础按平衡二叉树来组织数据,即:对每一个LUN建一个平衡二叉树,用LBA作为平衡二叉树的索引关键字.在查询二级读Cache时,首先根据LUN的编号找到对应的平衡二叉树的根节点,然后搜索平衡二叉树找到对应的节点.平衡二叉树查询、添加、删除算法的复杂度为$0(\log_2^N)$,效率较高.

4.3 实验评估

4.3.1 仿真测试系统的建立

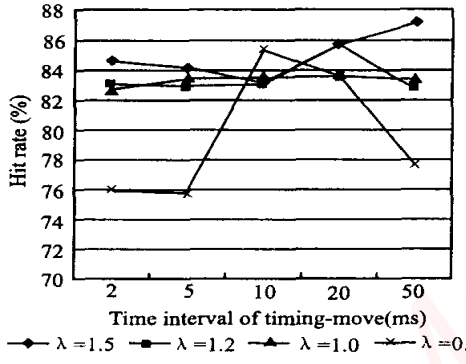
为了评价整个读Cache系统,建立了一个小型RAID系统,共分3个LUN,每个LUN容量为800M,RAID的读Cache一共48M,其中一级Cache为8M.同时还设计了一个仿真的输入系统,该系统通过模拟20个用户,每个用户连续输入大小为8K的请求12000个,这些请求的LBA值是半随

机性的(既保证请求的空间局部性, 又保证一定的随机性). 不同的用户输入的请求完全相同, 但输入开始的时间不一致, 20 个用户输入开始的时间满足泊松分布. 一般认为, 对相同数据块的请求间隔的时间越短, 则时间局部性越强. 因此, 泊松参数 λ 值越大则说明请求的时间局部性越强.



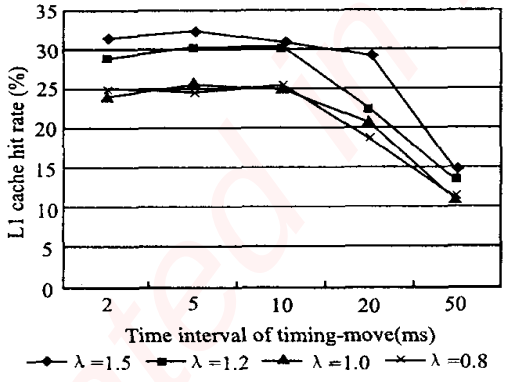
(a) 定时搬移策略的响应时间

(a) Response time of timing-move strategy



(b) 定时搬移策略的 Cache 命中率

(b) Cache hit rate of timing-move strategy



(c) 定时搬移策略一级 Cache 的命中率

(c) L I cache hit rate of timing-move strategy

图 4

Fig. 4

通过仿真系统, 测试了 RA ID 系统对 240000 个请求的响应时间、读 Cache 总的命中率以及一级 Cache 的命中率. 在测试中选取了不同的搬移时间间隔对定时搬移策略进行测试, 其中定时搬移策略的时间间隔分别为 2ms、5ms、10ms、20ms、50ms, 搬移淘汰的阈值为 70%; 同时还测试了淘汰搬移策略的性能, 并把两者进行对比.

4.3.2 测试数据分析

图 4(a)、(b)、(c) 分别给出了在 λ= 1.5、1.2、1.0、0.8 时采用不同的时间间隔进行定时搬移的响应时间、Cache 的命中率及一级 Cache 的命中率. 从图 4(a) 中可以看出: λ 越小即访问的时间局部性越弱, 其响应时间越长, 系统的吞吐量越小; 并且在 λ 较小时, 不同搬移时间间隔的性能差异也变大, 因此, 在不同的条件下, 如何确定最佳的搬移频率是需要进一步研究的内容. Cache 总的命中率及一级 Cache 的命中率也与访问的时间局部性紧密相关, 局部性越强, 其性能越好, 这一点在图 4(c) 一级 Cache 的命中率的统计上表现得尤为明显. 这证明定时搬移策略较好地满足了时间局部性的要求.

表 1 定时搬移与淘汰搬移比较 (其中定时搬移是取平均值)

Table 1 Contrasting performance of timing-move with replace-move (Using mean for timing-move)

		λ= 1.5	λ= 1.2	λ= 1.0	λ= 0.8
Response time(s)	Replace-move	390	385	403	397
	Timing-move	346	353	355	394
Cache hit rate (%)	Replace-move	84.1	83.4	81.6	82.5
	Timing-move	84.9	83.4	83.3	79.6
L1 Cache hit rate (%)	Replace-move	5.3	20.3	11.5	14.4
	Timing-move	27.7	25.0	21.2	21.0

为了说明问题, 把相同输入条件下采用淘汰搬移策略与定时搬移策略的响应时间、Cache 命中率及一级 Cache 命中率进行对比(表 1). 从表 1 中可以看出, 淘汰搬移的性能与 λ 值即时间的局部性没有很大的关系, 它可以看成是二级 Cache 的延伸. 虽然, 在整体上两种策略 Cache 的命中率差别不大, 但是其响应时间差别较大. 其主要原因是: 一方面一级 Cache 的命中率差别较大, 定时搬移策略的一级 Cache 命中率要显著高于淘汰搬移; 另一方面淘汰搬移策略的"搬移"操作占用了处理请求的时间.

5 结 论

本文根据 RA ID 的工作情况以及 Cache 的空间、时间局部性原理设计了一种新的基于 RA ID 的 Cache 系统. 本系统在 Cache 的全局配置上采取了预留与动态分配相结合的方法, 并建立了分配模型, 能够根据 RA ID 的用户访问情况确定预留与动态分配的最佳比例. 在读 Cache 的设计上, 本系统采用两级结构, 通过一级 Cache 实现时间的局部性; 二级 Cache 实现空间的局部性. 通过定时搬移并按阈值淘汰的策略完成二级 Cache 数据向一级 Cache 搬迁. 从仿真实验中可以看出, 采用定时搬移策略的 Cache 系统不仅较好地实现了时间的局部性, 而且与淘汰搬移策略相比系统的性能有较大的提高.

本系统可以用于事务处理等时间局部性较强的领域. 为了更好地提高效能, 还需要进行以下两个方面的研究: (1) 如

何准确地度量系统在忙、闲时的访问速率; (2) 如何从大量的访问数据中抽象出时间局部性的强弱.

References:

1 Thomasian A, Menon J. Performance analysis of RAID5 disk arrays with a vacationing server model for rebuild mode operation [A]. Data Engineering [C]. 1994. Proc. 10th International Conference, Feb 1994, 111-119.

2 Thomasian A. Priority queueing in RAID5 disk arrays with an NVS Cache, modeling analysis and simulation of computer and telecommunication systems [C]. 1995. Proc. of the Third International Workshop on , Jan 1995 , 168-172.

3 Menon J, Cortney J. The Architecture of a fault-tolerant Cached RAID controller. Computer architecture[C]. 1993. Proc. of the 20th Annual international Symposium on May 1993, 76-86.

4 Biswas P, Ramskrishnan K, Towsley D. Trace-driven analysis of caching policies for disks[C]. Proc. 1993 ACM Sigmetrics Conf. Measurement and Modeling of Computer Systems, May 1993, 13-23.

5 Miao Jun-hai, Zhu Lan-juan, Wu Zhi-ming. Design and implementation of the cache in RAID [J]. Microcomputer Applications, 2001(4): 29-31.

6 Chen Yun, Yang Gen-ke, Wu Zhi-ming. The application of two-level cache in RAID system [C]. Proc. of the 4# World Congress on Intelligent Control and Automation, June 2002, 1328-1332.

7 Anujan Varma, Quinn Jacobson. Destage algorithms for disk arrays with nonvolatile caches [J]. IEEE Transactions on Computers, Fevruary 1998, 228-235.

8 Jung-Hoon Lee, Jang-Soo Lee, Shin-Dug Kim, A new cache architecture based on temporal and spatial locality[J]. Journal of Systems Architecture Dec 2000 1451-1467.

9 John L. Hennessy, David A. Patterson, Computer architecture: a quantitative approach, Third edition [M]. Elsevier Science Pte Ltd. 2003, 392-448.

附中文参考文献:

5 缪军海, 朱兰娟, 吴智铭. RAID 中 Cache 的设计与实现[J]. 微型电脑应用, 2001(4): 29-31.

嵌入式资源免费下载

总线协议：

1. 基于 PCIe 驱动程序的数据传输卡 DMA 传输
2. 基于 PCIe 总线协议的设备驱动开发
3. CANopen 协议介绍
4. 基于 PXI 总线 RS422 数据通信卡 WDM 驱动程序设计
5. FPGA 实现 PCIe 总线 DMA 设计
6. PCI Express 协议实现与验证
7. VPX 总线技术及其实现
8. 基于 Xilinx FPGA 的 PCIE 接口实现
9. 基于 PCI 总线的 GPS 授时卡设计
10. 基于 CPCI 标准的 6U 信号处理平台的设计
11. USB30 电路保护
12. USB30 协议分析与框架设计
13. USB 30 中的 CRC 校验原理及实现
14. 基于 CPLD 的 UART 设计
15. IPMI 在 VPX 系统中的应用与设计
16. 基于 CPCI 总线的 PMC 载板设计
17. 基于 VPX 总线的工件台运动控制系统研究与开发
18. PCI Express 流控机制的研究与实现
19. UART16C554 的设计
20. 基于 VPX 的高性能计算机设计
21. 基于 CAN 总线技术的嵌入式网关设计
22. Visual C 串行通讯控件使用方法与技巧的研究
23. IEEE1588 精密时钟同步关键技术研究
24. GPS 信号发生器射频模块的一种实现方案
25. 基于 CPCI 接口的视频采集卡的设计
26. 基于 VPX 的 3U 信号处理平台的设计
27. 基于 PCI Express 总线 1394b 网络传输系统 WDM 驱动设计
28. AT89C52 单片机与 ARINC429 航空总线接口设计
29. 基于 CPCI 总线多 DSP 系统的高速主机接口设计
30. 总线协议中的 CRC 及其在 SATA 通信技术中的应用
31. 基于 FPGA 的 SATA 硬盘加解密控制器设计
32. Modbus 协议在串口通讯中的研究及应用

VxWorks：

1. 基于 VxWorks 的多任务程序设计
2. 基于 VxWorks 的数据采集存储装置设计
3. Flash 文件系统分析及其在 VxWorks 中的实现
4. VxWorks 多任务编程中的异常研究
5. VxWorks 应用技巧两例
6. 一种基于 VxWorks 的飞行仿真实时管理系统
7. 在 VxWorks 系统中使用 TrueType 字库
8. 基于 FreeType 的 VxWorks 中文显示方案
9. 基于 Tilcon 的 VxWorks 简单动画开发
10. 基于 Tilcon 的某武器显控系统界面设计
11. 基于 Tilcon 的综合导航信息处理装置界面设计
12. VxWorks 的内存配置和管理
13. 基于 VxWorks 系统的 PCI 配置与应用
14. 基于 MPC8270 的 VxWorks BSP 的移植
15. Bootrom 功能改进经验谈
16. 基于 VxWorks 嵌入式系统的中文平台研究与实现
17. VxBus 的 A429 接口驱动
18. 基于 VxBus 和 MPC8569E 千兆网驱动开发和实现
19. 一种基于 vxBus 的 PPC 与 FPGA 高速互联的驱动设计方法
20. 基于 VxBus 的设备驱动开发
21. 基于 VxBus 的驱动程序架构分析
22. 基于 VxBus 的高速数据采集卡驱动程序开发

Linux：

1. Linux 程序设计第三版及源代码
2. NAND FLASH 文件系统的设计与实现
3. 多通道串行通信设备的 Linux 驱动程序实现
4. Zsh 开发指南-数组
5. 常用 GDB 命令中文速览
6. 嵌入式 C 进阶之道
7. Linux 串口编程实例
8. 基于 Yocto Project 的嵌入式应用设计
9. Android 应用的反编译
10. 基于 Android 行为的加密应用系统研究
11. 嵌入式 Linux 系统移植步步通

12. 嵌入式 CC++语言精华文章集锦
13. 基于 Linux 的高性能服务器端的设计与研究
14. S3C6410 移植 Android 内核
15. Android 开发指南中文版
16. 图解 Linux 操作系统架构设计与实现原理（第二版）
17. 如何在 Ubuntu 和 Linux Mint 下轻松升级 Linux 内核
18. Android 简单 mp3 播放器源码
19. 嵌入式 Linux 系统实时性的研究
20. Android 嵌入式系统架构及内核浅析
21. 基于嵌入式 Linux 操作系统内核实时性的改进方法研究
22. Linux TCP IP 协议详解
23. Linux 桌面环境下内存去重技术的研究与实现
24. 掌握 Android 7.0 新增特性 Quick Settings

Windows CE：

1. Windows CE.NET 下 YAFFS 文件系统 NAND Flash 驱动程序设计
2. Windows CE 的 CAN 总线驱动程序设计
3. 基于 Windows CE.NET 的 ADC 驱动程序实现与应用的研究
4. 基于 Windows CE.NET 平台的串行通信实现
5. 基于 Windows CE.NET 下的 GPRS 模块的研究与开发
6. win2k 下 NTFS 分区用 ntldr 加载进 dos 源代码
7. Windows 下的 USB 设备驱动程序开发
8. WinCE 的大容量程控数据传输解决方案设计
9. WinCE6.0 安装开发详解
10. DOS 下仿 Windows 的自带计算器程序 C 源码
11. G726 局域网语音通话程序和源代码
12. WinCE 主板加载第三方驱动程序的方法
13. WinCE 下的注册表编辑程序和源代码
14. WinCE 串口通信源代码
15. WINCE 的 SD 卡程序[可实现读写的源码]
16. 基于 WinCE 的 BootLoader 研究

PowerPC：

1. Freescale MPC8536 开发板原理图

2. 基于 MPC8548E 的固件设计
3. 基于 MPC8548E 的嵌入式数据处理系统设计
4. 基于 PowerPC 嵌入式网络通信平台的实现
5. PowerPC 在车辆显控系统中的应用
6. 基于 PowerPC 的单板计算机的设计
7. 用 PowerPC860 实现 FPGA 配置
8. 基于 MPC8247 嵌入式电力交换系统的设计与实现
9. 基于设备树的 MPC8247 嵌入式 Linux 系统开发
10. 基于 MPC8313E 嵌入式系统 UBoot 的移植
11. 基于 PowerPC 处理器 SMP 系统的 UBoot 移植

ARM：

1. 基于 DiskOnChip 2000 的驱动程序设计及应用
2. 基于 ARM 体系的 PC-104 总线设计
3. 基于 ARM 的嵌入式系统中断处理机制研究
4. 设计 ARM 的中断处理
5. 基于 ARM 的数据采集系统并行总线的驱动设计
6. S3C2410 下的 TFT LCD 驱动源码
7. STM32 SD 卡移植 FATFS 文件系统源码
8. STM32 ADC 多通道源码
9. ARM Linux 在 EP7312 上的移植
10. ARM 经典 300 问
11. 基于 S5PV210 的频谱监测设备嵌入式系统设计与实现
12. Uboot 中 start.S 源码的指令级的详尽解析
13. 基于 ARM9 的嵌入式 Zigbee 网关设计与实现
14. 基于 S3C6410 处理器的嵌入式 Linux 系统移植
15. CortexA8 平台的μC-OSⅡ及 LwIP 协议栈的移植与实现

Hardware：

1. DSP 电源的典型设计
2. 高频脉冲电源设计
3. 电源的综合保护设计
4. 任意波形电源的设计
5. 高速 PCB 信号完整性分析及应用

6. DM642 高速图像采集系统的电磁干扰设计
7. 使用 COMExpress Nano 工控板实现 IP 调度设备
8. 基于 COM Express 架构的数据记录仪的设计与实现
9. 基于 COM Express 的信号系统逻辑运算单元设计
10. 基于 COM Express 的回波预处理模块设计
11. 基于 X86 平台的简单多任务内核的分析与实现
12. 基于 UEFI Shell 的 PreOS Application 的开发与研究
13. 基于 UEFI 固件的恶意代码防范技术研究