盘阵中基于平衡超图划分的自适应请求并发与负载平衡策略

刘 军, 杨学军, 王俊伟, 唐玉华
(国防科大计算机学院 604 教研室, 湖南长沙 410073)

摘 要: 作为一种并行系统, 盘阵性能极大地依赖于设备间负载平衡; 为了减少执行时间, 单个 I/O 请求需要通过多个设备并发完成, 负载平衡并不意味着请求并发, 而请求并发也不能保证盘阵负载平衡. 因此, 必须将二者结合起来, 才能使盘阵性能得到更好的优化. 为此本文提出基于平衡超图划分的自适应数据分布策略, 并提出了两个目标函数同时进行请求并发与负载平衡的控制, 根据 I/O 访问模式优化盘阵性能. 根据两种 I/O benchmark 合成负载进行模拟试验表明, 该策略比传统的单方面自适应负载平衡策略或基于超图的请求并发策略效果都有一定优化.

关键词: 盘阵; 请求并发; 负载平衡; 平衡超图划分
中图分类号: TP302.8、TP303 文献标识码: A 文章编号: 0372-2112 (2005) 04-0735-07

An Adaptive Strategy Based on Balanced Hypergraph-Partition for Both Load Balance and Intra-request Concurrency in Disk Array System

LIU Jun, YANG Xue-jun, WANG Jun-wei, TANG Yu-hua
(*Section 604, School of Computer, National University of Defense Technology, Changsha, Hunan 410073, China*)

Abstract: Disk array is a high performance storage system for exploiting intra-request concurrence and inter-request parallelism. Load balance among disks is necessary for better inter-request parallelism, and an I/O request should be served by multiple disks for intra-request concurrence. This can be achieved for optimal data placement. But file access pattern is changing. System should redistribute data blocks adaptively. In this paper, we present a strategy of balanced hypergraph-based adaptive data placement, and give two objective functions for the optimization of both load balance and intra-request concurrence. We have done lots of simulations to evaluate our method and compare it with traditional optimization strategies such as adaptive load-balance and hyper-graph placement for intra-request concurrency. The result manifests that our strategy is a useful one.

Key words: disk array; intra-request concurrency; load balanced; balanced hypergraph-partition

1 引言

盘阵中负载平衡是指单位时间内各设备处理请求的数目基本相同; 而请求内部并发性(Intra-request concurrency)是指一个大 I/O 请求能通过多个设备并行执行, 简称请求并发. I/O 是计算机性能瓶颈. 盘阵是通过请求内部并发和请求间并发(Inter-request concurrency)获取高性能[13], 这涉及到请求并发与负载平衡问题.

作为一种并行系统, 盘阵性能极大地依赖于负载平衡[15]; 而请求并发决定了 I/O 请求的快速执行. 负载平衡决定了盘阵整体上请求处理的吞吐率和加速比. 好的负载平衡能减小设备上请求队列长度, 使系统中设备的利用率基本相同, 从而性能得到充分发挥. 然而负载平衡只能根据统计一大段时间内设备的负载情况进行判断, 时间上是一种宏观的概念; 微观上具体到请求需要设备并发执行, 减少请求的响应时间. 宏观上的负载平衡并不能说明微观上的请求并发, 而微观上的请求个体的并发同样并不能保证设备的负载平衡.

本研究首先对两种 I/O benchmark: iostone[16] 和 postmark[8] 进行分析. iostone 是一种传统的文件系统 benchmark, 其中文件的大小差别很大, 256 字节的文件占 32.3%, 长度为 512、1024、2048 的文件各占 16.2%, 4KB 和 8KB 的文件各占 8.1%, 16KB 的文件占 2.0%, 64KB 和 32KB 的文件各占 0.5%; I/O 读与写请求的比例为 2:1, 且每个 I/O 请求都是对整个文件进行操作, 请求所访问的具体文件是随机的. postmark 是有关电子邮件、网络新闻和基于 web 的商业应用等分布式事务的 benchmark, 其操作包括对文件的 create、delete、read 和 append 等, 每个事务包含操作队(create 或 delete, 和 read 或 append), 一次测试包含 1000 或 2000 个文件, 执行 50000 或 100000 个事务, 文

收稿日期: 2003-12-01; 修回日期: 2004-10-20
基金项目: 国家杰出青年科学基金(No. 69825104); 国家 863 项目(No. 2002AA1Z2101)

件大小是最小值(默认 500 字节) 至最大值(默认 10000 字节)之间的一个随机数,(当然这些参数可以用户定义), 每次请求操作的文件都是随机的, read 和 create 都一次性操作整个文件.

这些分析表明,系统许多 I/O 请求都是访问一些相同的数据块、且此类 I/O 访问的频率特别高. 进行优化时,必须让这些请求通过盘阵的多个设备并发完成,并且使盘阵各设备的负载平衡. 这通过将数据在盘阵的各设备上进行合理分布来实现. 然而最优的数据分布是 NP 完全问题,但是可以根据请求序列特点,给出一些自适应算法.

已有的一些关于自适应负载平衡和请求并发的研究. 负载平衡研究如磁盘冷却算法[13] 和文献[1] 中的 Hippodrome 工具等忽略了请求并发;请求并发的研究如基于最大切分的图划分[14] 和基于超图的数据分布[9] 等并未考虑负载平衡的问题. 因此这些工作不能使盘阵发挥出最大潜能.

结合以往相关工作,本文对负载平衡与请求并发相结合的盘阵优化策略进行了研究,主要创新工作是: ①提出了基于平衡超图划分的自适应数据分布策略,该策略首先将 I/O 访问序列形成超图,然后采用平衡超图划分的算法,将负载平衡与请求并发技术同时考虑,进行系统优化; ②在进行平衡超图划分的控制过程中,本研究提出了两个新的目标函数,分别对请求并发和负载平衡进行控制. 本研究提出的目标函数分布与请求并发和负载平衡保持单调关系,目标函数的值越小,意味着系统越优化; ③最后,本研究进行了模拟试验. 试验的 I/O 请求负载是根据 iostone 和 postmark 合成的. 数据的原始分布是 round-robin 和以文件为单位两种. 试验表明,针对不同的 I/O 访问模式和不同的初始分布,如果仅仅进行负载平衡的优化[13] 或者仅仅进行请求并发的优化[9], 一定的优化效果都只能针对特殊分布,而平衡超图划分策略却能针对多种情况产生更好的优化.

2 相关工作

请求并发和负载平衡一直是盘阵性能优化的研究热点之一[6].

请求并发方面,许多研究是针对应用中数据结构的多维特点,采用映射函数如 CMD(Coordinate modulo declustering) 函数[9] 将数据分散在各设备上,实现请求内部的并发性. 但是基于函数的分布是静态的. 实际系统中数据访问特性是变化的,静态函数分布不能很好地利用当前系统中已知变化信息如系统的请求访问序列、各数据块的访问情况等,进行自适应优化. 文献[14] 提出的基于相似图和文献[9] 提出基于超图的数据分布,自适应提高数据分布的并发性,但他们没有考虑盘阵设备负载平衡.

负载平衡方面,文献[3] 提出 Bubba 算法,在将关系数据库中负载重的关系存放在负载轻的设备上. 但是请求是动态的,随着数据库记录加入和删除,原有的关系会出现负载不平衡. 静态分布不能为动态访问提供负载平衡. 文献[1] 的 Hippodrome 工具、EMC 的 Symmetrix[5] 和 HP SureStore E 盘阵[7], 讨论了数据的自适应重构问题,它们监测各数据单元的利用率情况,采用一个阀值进行判断,触发数据迁移,达到负载平衡. 但这些负载平衡方面的工作都没考虑请求内部并发,只在宏观上实现负载平衡.

一个高效的盘阵系统,既要负载平衡也要请求并发. 也就是说,系统管理优化软件必须做到: ①通过监控负载情况,进行自适应负载平衡; ②为了负载平衡而进行数据块重构时,不能减少请求的并发性; ③对于一个平衡的系统,系统管理优化软件还需要监控数据块间的并发关系,进行自适应调整,增加请求内部的并发性.

系统的 I/O 负载是可以通过一定方式分析和预测的[4], 而请求的访问模式也可以通过一定工具进行分析,且 I/O 一般是有特点的[1]; 文献[11] 和文[2] 提出的一些在线数据迁移技术和试验表明,虽然自适应数据重构会带来一些负载,但是这些额外负载可以通过一定方式将影响控制在很小的范围内. 因此自适应优化对提高盘阵性能是有意义的.

3 问题描述

用图划分方法来进行数据分布,是经常采用的方式[9]. 本研究采用平衡超图划分的方法来描述请求的数据分布优化.

3.1 平衡超图

定义一: 超图(Hypergraph) $H=(V, Nets)$, V 是顶点集, $Nets$ 是超边的集合. 每条超边 $n_j \in Nets$ 包含一些顶点,其大小就是包含的顶点数,计为 $|n_j|$.

定义二: 给定超图中任意超边 n_j 加权为 w_j, 任意顶点 v 加权为 $lb(v)$. 如果将超图中这些顶点划分为 N 个子集 $\prod_N = \{V_1, V_2, \cdots, V_N\}$, 任给顶点集 V_i, 令其权 lD_i 为该顶点集中所有顶点的权之和,即 $lD_i = \sum_{v \in V_i} lb(v)$; 在该划分中,对于任意超边 n_j, 用 $n_j(k) \subset n_j$ 表示超边 n_j 划分在组 V_k 的顶点集,也就是说 $n_j(k) = n_j \cap V_k$, $k = 1, 2, \cdots, N$; 并令 n_j 划分在 $\prod_N$ 中的最大顶点数 δ_j^{max}, 即 $\delta_j^{max} = \max_{1 \leqslant k \leqslant N}\{|n_j(k)|\}$, 且令相应的最小顶点数为 δ_j^{min}, 即 $\delta_j^{min} = \min_{1 \leqslant k \leqslant N}\{|n_j(k)|\}$. 对于任意超边 n_j, 如果 δ_j^{max} 与 δ_j^{min} 间差值不大于 1,且 $lD_1 = lD_2 = \cdots = lD_N$, 则称该划分 $\prod_N$ 为平衡超图划分.

超图的定义(定义一) 在文献[9] 中讨论请求并发时已经提出. 有别于此,为了同时描述请求并发和负载平衡,本研究提出了平衡超图的概念(定义二).

3.2 数据分布

如果将盘阵系统数据单元 SU(Stripe Unit) 的集合作为顶点集 V, 而将每个请求 q_i 看成是一条超边 n_j(一个请求包含多个数据单元),这些请求的集合就构成超边的集合 $Nets$, 则盘阵的请求信息与数据分布可看成是一个超图. 每条超边的权定义成此类请求的频率 $w_j = f(q_i)$. 每个顶点的权 lb 定义成该顶点访问频率(即该结点的负载), 将盘阵中特定设备上分布的数据块集看成是划分 $\prod_N$ 中某顶点集 V_i, 则盘阵数据分布优化问题转换成平衡超图划分问题.

事实上,自适应数据分布几乎不可能达到完全平衡超图划分,我们只能采用一定的目标函数进行控制、判断和优化. 因为对于任一超边 n_j, δ_j^{max} 与 δ_j^{min} 越接近,请求并发性越好;

而每个设备的负载 lD 越相等, 意味着设备间负载越平衡, 所以我们分别用不同的目标函数来描述.

3.2.1 请求并发目标函数　　为了使请求并发, 必须根据目标函数对每个请求进行判断. 从定义二可知, 任意超边 n_j 的目标函数: $c\prod_N(n_j)=\delta_j^{max}-\delta_j^{min}$, $c\prod_N(n_j)$ 越小, 说明请求 n_j 越能并发, 因此超图划分的目标函数可表示为:

$$c\prod_N(H)=\sum_{n_j\in N(H)} w_j c\prod_N(n_j)$$
$$=\sum_{n_j\in N(H)} w_j\Big(\max_{1\leqslant k\leqslant N}\{|n_j(k)|\}-\min_{1\leqslant k\leqslant N}\{|n_j(k)|\}\Big) \quad (1)$$

超边 n_j 的目标函数 $c\prod_N(n_j)$ 与请求并发之间保持单调变化关系, 因此整个超图划分的目标函数 $C_{\prod N}$ 也就能在整体上保持这种特性. 也就是说 $C_{\prod N}$ 越小, 请求并发性越好, 是一种单调关系.

我们提出的目标函数, 与以往的处理方式是不同的. 在传统方法中, 基于相似图的方法[14]中, 目标函数与请求并发性不保持单调关系[9]; 虽然文献[9]提出了基于超图的分布, 对于任意超边 j 令 $\delta_j^{pt}=\lceil|n_j|/N\rceil$, 该文采用 $c_{\prod_N}(n_j)=\delta_j^{max}-\delta_j^{pt}$ 的目标函数, 对超边的并发性进行判断和控制. 该方式不能使并发达到最优, 比如一个请求访问 4 个数据块, 而盘阵有 3 个设备, 如果在两个设备上放置两块, 一个设备空闲, 则该判断方式的目标函数值为 0, 认为这种分布已经达到最优, 事实上这种分布对请求的并发性和设备的负载平衡性都不好; 而采用等式(1)则可以判断这种方式不是最优的, 因为 $c_{\prod_N}(n_j)=2>1$.

3.2.2 负载平衡目标函数　　负载平衡是整体上的概念, 需从宏观上考虑. 系统总负载 TL: $TL=\sum_{i=1}^{N} lD_i$, 令盘阵中负载最轻的设备上负载为 lD_{min}, 即 $lD_{min}=min_{1\leqslant k\leqslant N}\{lD_i\}$; 而负载最重的设备负载为 lD_{max}, 即 $lD_{max}=max_{1\leqslant k\leqslant N}\{lD_i\}$, 负载偏差 DLV:

$$DLV=lD_{max}-lD_{min} \quad (2)$$

负载平衡的目标是使设备负载偏差 DLV 最小, 因为 DLV 越小, 与设备间负载越相等或者各设备上负载越接近平均值 TL/N, 是一种单调关系.

目标函数等式(2)与传统的采用类似于均方差的方式判断负载平衡[13]是不同的. 所谓均方差判断是计算 $DLV=\sum_{i=1}^{N}(TL/N-lD_i)^2$, 通过改进数据分布使 DLV 最小, 该目标函数不是判断负载平衡最好的方式, 也就是说该目标函数与负载平衡不保持单调关系. 比如, 一个盘阵有 3 个设备, 各设备的负载情况①是: 5、5、1, 此时 DLV 是 10.7; 如果设备的负载情况②是: 6、3、2, 其 DLV 是 8.7. 均方差判断方法表明情况②比情况①负载更平衡, 但事实上从分布①向分布②转变时并不能对负载平衡进行优化, 反而负载更不平衡, 因为原来情况①中负载重的设备(disk1)在情况②中负载更重. 采用等式(2)则可以避免这些问题.

3.3 信息记录

式(1)和式(2)共同构成我们算法的目标函数. 如果使得 DLV 和 $C_{\prod N}$ 同时最小, 则负载平衡和请求并发同时满足. 为了能很好的使用平衡超图划分方法, 我们需对盘阵请求进行记录.

对盘阵的所有请求都需要经过盘阵的管理软件(盘阵驱动程序), 才能将请求转发到相应设备. 盘阵管理优化软件记录这些请求的访问情况. 自适应策略是根据当前系统记录的一些信息形成超图, 采用平衡超图划分策略, 对数据分布进行调整. 记录的信息必须包含负载和请求信息. 记录的信息包括请求号、请求对应的设备块号、和该请求的频率、设备上各结点上数据块的负载等统计信息.

4　平衡超图划分实现算法

为了同时考虑负载平衡与请求并发, 我们必须根据目标函数对系统的监测和记录情况进行判断, 并根据一定算法对现有数据分布进行重构.

4.1 算法

数据分布优化是在当前分布的基础上根据一定的策略进行自适应调整. 在我们的算法中, 首先形成 action 序列, 然后一次完成这些 action. 记录 action 类型是:

```
struct action_type {
  block_no; // 顶点号, 设备上数据块块号
  lb; // 顶点负载, 数据块负载
  origin_dik_no; // 数据块的源设备号
  target_disk_no; // 目的设备号
} action_type
```

在算法中, ACTION 函数是:

```
ACTION(u: 顶点, s: 源设备, t: 目的设备)
{
  action.block_no= u 的块号;
  action.lb= lb(u);
  action.origin_disk_no= s;
  action.target_disk_no= t;
  将 action 添加到 actions_to_do 队列中;
  V[s]= V[s]-{u}; // 源顶点集减去 u
  V[t]= V[t]+{u}; // 目的顶点集增加 u
}
```

ACTION 函数是形成将顶点 u 从源顶点集 $V[s]$ 迁移到目的顶点集 $V[t]$ 的 action; 将 action 添加到 actions_to_do 队列中时, 如果队列中已经有了关于该顶点的项, 那么就修改相关域 origin_disk_no 和 target_disk_no, 如果 origin_disk_no 与 target_disk_no 相等, 则将该 action 删除.

4.1.1 平衡超图划分算法　　对盘阵的 I/O 请求既能并发又能使盘阵负载平衡见算法 4-1. 算法主要包括两步, 第一步是进行请求并发的优化工作, 根据等式(1)求出 $Cost\_Par$ (即 $C_{\prod N}$), 如果 $Cost\_Par$ 大于一给定正数, 则进行请求并发的数据重构; 第二步是进行负载平衡的优化工作, TL 是系统总负载, AL 是设备的平均负载, $ReLD[i]$ 是设备 i 的负载偏差, 也就是说 $ReLD[i]=lD[i]-AL$. 根据等式(2)求出 DLV, 如果 DLV 大于一个数, 则进行负载平衡的数据重构, 同时保持

Cost_ Par 不增加.

```
输入:
  数组 V[N]: N 个互不相交的顶点集
  数组 lb[N][]: 顶点集中各顶点负载
  数组 Nets[]: 超边集合
  数组 f[]: 不同超边的负载
变量:
  数组 ID[N]: 不同顶点集的负载
  TL: 负载总和
  AL: 设备的平均负载
  ReID[N]: 各顶点集需迁移的负载量
  Cost_par: 当前划分的目标函数值
  DLV: 设备负载偏差和
  M: 所有 顶点的集合
  action : 临时动作
输出:
  actions_ to do: 需要进行重构的动作序列
算法:
  actions_ to_ do= empty;
  第一步: 进行并发的数据重构
    计算 Cost_ Par; (一给定正数)
    if Cost_ Par> Epsilon1
    Partition_ Refine(Nets, V);
    //进行数据并发分布的优化
  第二步:
    for( i= 0; i< N; i+ + )
      计算各设备的负载 ID[i];
    计算 TL、AL、ReLD[N]、DLV;
    M= 所有顶点的集合;
    while DLV> Epsilon 2(一给定正数){
    //从负载高的顶点组选择顶点移动到负载低的组
    对于负载高的顶点集中所有顶点 u, 且 u∈M{
      从负载高的顶点集 s 选取顶点 u;
      并选出负载轻的顶点集 t;
      计算 gain= move_ gain(u, s, t);
      if (lb(u)≤ReLD[u])&&
        (lb(u)≤- ReLD[t])&&(gain≥0)) {
        ACTION(u, s, t);
        重新计算 ReLD 和 DLV;
      }
      M= M- {u};
      If(M 为空或者 DLV≤Epsilon2) goto 第三步
      }
    }//end of while
  第三步:
    完成 actions_ to_ do 队列中所有操作.
```

算法 4- 1 平衡超图划分的算法

算法 4- 1 中, *partition_ Refine* 函数进行数据并发优化, 其算法见算法 4- 3. 这里关于负载平衡重点说明几点:

第一, 通过计算各结点的负载, 得出各设备的负载偏移. 对于负载重的设备来说, 偏移的负载量就是在负载平衡过程中需要调整出去的负载量, 而对于负载轻的设备来说, 相应的负载偏移是需要加入的负载量. 因此在选取要调整的数据块时必须适合这些负载偏移. 这样作有利于负载平衡的快速收敛.

第二, 选取需要移动的数据块时还需考察数据块的移动是否能满足并发性要求. 这是通过 *move_ gain* 函数计算, 其计算算法见算法 4- 2.

第三, 选择出合适的数据块后, 调用 *ACTION* 函数形成 *action* 加入到 *actios_ to_ do* 队列中, 并重新计算 DLV. 如果 DLV 已经符合要求, 则算法结束.

算法结束后, 完成 *actios_ to_ do* 队列中所有 *action*.

算法中引入 M 顶点集, 是防止已经考察过的数据块重复考察.

4.1.2 数据移动的利润获取计算 对请求并发性来说, 将数据块从一个设备移动到另一个设备时有并发性损失 cost, 或者有好处 benefit, 一定要进行利润 gain 的计算, 考察数据移动对并发性的影响. 计算方法类似于[9].

```
move_ gain(u: 顶点; s: 源顶点集号; t: 目的顶点集号
{
  gain= 0; benefit= 0; cost= 0; //初始化
  for each n_j∈ nets[u]{
    δ_j= |n_j|/N
    if (|n_j(s)|≤δ_j) ||(|n_j(t)|≥δ_j)
      cost+ = f(n_j);
    else benefit + = f(n_j);
    }
    gain= benefit- cost;
    return gain;
}
```

算法 4- 2 负载移动带来的好处

数据块 u 从源设备 s 移动到目的设备 t, 对应到超图中是将顶点 u 从顶点集 $V[s]$ 移动到 $V[t]$: ①但是如果有超边 n_j 分布在 $V[t]$ 的顶点数 $|n_j(t)|$ 已经不小于平均的数目 δ_j, 或者分布在 $V[s]$ 的顶点数 $|n_j(s)|$ 已经不大于平均的数目 δ_j, 则此次移动带来的开销 cost 是 $f(n_j)$; ②如果有超边 n_j 分布在 $V[s]$ 的顶点数 $|n_j(s)|$ 超过平均的数目 δ_j, 且目的结点 $V[t]$ 上分布的顶点数小于 δ_j, 显然此次移动能带来的好处 benefit 是 $f(n_j)$; benefit 与 cost 相减既得出移动收获 gain. 其余的情况认为这种移动既不能带来 cost, 也不能带来 benefit.

当然, 从直观上看, 也还存在那种情况: 一条超边是另一条超边子集, 简单的将一个数据块从一个设备移动到另一个设备上, 按照算法 4- 2 计算不会获取利润, 但是从这两条超边的分布在不同设备上的数据块进行交换, 则可能带来利润. 我们从 I/O stone 和 Postmark benchmark 的分析中发现, 这种情况不存在. 因此, 我们的算法没有考虑这种情况.

4.1.3 并发的数据调整算法 有了 move_ gain 函数后数据并发的调整就简单多了. 数据移动算法如 4- 3. 算法步骤如下: 如果当前的数据分布目标函数值 *Cost_ Par* 太大, 则对所有的顶点进行考察, 如果移动有利可图, 则将顶点进行移动.

数据需要经过多遍搜索才能使 *Cost_Par* 符合要求.

```
partition_Refin (Nets: 超边集;  V: 顶点集)
{
    while Cost_Par> Epsilon1 (一给定正数){
      M= 所有顶点的集合;
      for ( s= 0; s< N; t+ + , t≠s){
        从 V[ s] 中选取顶点 u 且 u∈M,
        for ( t= 0; t< N; t+ + , t≠s){
          计算 gain= move_ gain (u, s, t);
          if (gain> 0) {
            ACTION ( U, S, T);
            //将顶点 u 从 V[ s] 迁移到 V[ t]
          break;
          }
        }
      重新计算 Cost_par,
      M = M- { u};
      if (M 为空域者 Cost_ Par≤Epsilonl)
              break;
      }
    }
}
```

算法 4- 3 负载移动是请求并发

4.2 基于请求的重构技术

数据的调整会对盘阵系统引起新的额外负载. 以往的数据重构与当前用户请求是分离的, 这样会在负载重构过程中引起一些新的负载. 我们提出基于用户请求的数据选取和调整, 使重构负载与用户请求相重叠, 减少负载平衡的 IO 开销.

文献[17] 的统计表明, 在文件系统中, 文件要么被打开进行只读, 要么被打开进行只写, 而被打开后即读又写的文件很少. 通过分析 I/O stone 和 postmark benchmark 的分析中, 虽然文件可能有读有写, 不能完全反映这种特性, 但是可以想象, 有的环境如多媒体服务器、新闻服务器和 email 服务器中, 文件一般只写一次, 此后都是多次读. 针对这种情况, 我们提出基于请求的方法是这样的: 对于如果需迁移的数据块为读数据块, 可以待服务用户请求时将数据读出后(节省了一次读操作), 再将其写入目的设备; 如果需迁移的是写数据块, 可以待服务用户发出写请求时将其重定向写入空闲的设备(无须额外操作).

5 模拟试验

为了验证本研究中基于平衡超图划分的数据分布算法对盘阵性能的优化效果, 我们通过模拟试验进行测试验证, 并同原来传统的仅仅进行负载平衡(磁盘冷却算法[13]) 、或请求并发(基于超图划分的优化[9]) 的优化相比较.

5.1 试验模型

我们编写了专门用于本研究的模拟器. 模拟器主要包括: 请求产生对象(RequestSynthesizer) 、盘阵的请求监控与信息记录对象(RequestRecorder) 、优化策略对象(Strategy) (如平衡超图划分) 、盘阵对象(DiskArray) 、磁盘设备部件(Disk). 请求产生器根据一定负载合成方式, 产生 I/O 请求, 向盘阵发出; 请求监控对象记录请求设备上各数据块的请求访问信息; 优化策略对象进行优化(如果采用平衡超图划分策略, 则将记录的信息形成超图, 对象根据等式①和②进行判断, 并根据算法 4 - 1 进行自适应数据重构); 盘阵对象内部有一个数据映射表, 记录数据块在物理磁盘上的寻址. 该映射表由盘阵与优化策略对象共同访问维护. 策略对象对数据分布进行重构时, 必须修改这个表. 盘阵根据这个表将上层的请求转发到相应设备. 磁盘构件完成盘阵调度的请求, 每个磁盘对象有自己的请求处理队列, 每个设备以 FCFS (First Come First Service) 服务队列中请求, 并返回结果, 不考虑请求合并.

模拟中, 只考虑磁盘操作的时间, 不考虑 cpu、cache 和总线传输的影响.

测试中模拟 6 个磁盘的 RAID0 盘阵, 盘阵的 SU 大小为 1K 字节. 模拟实验采用磁盘 Quantum Atlas 10K Ⅱ 73.4GB, 其参数如表 4- 1 所示. 模拟器中, 磁盘的请求服务时间包括寻道时间、旋转时间和数据传输时间. 其中数据传输时间是 SU 大小除以设备传输率; 旋转时间是 0~ 旋转一周的时间之间的一随机数; 寻道时间采用[10] 中的连续函数:

$$Seek(x)=\begin{cases}0, & \text{if } x=0\\ a\sqrt{x-1}+b(x-1)+c, & \text{if } x>0\end{cases}$$

而 a, b 和 c 由以下公式进行估计:

$a=(-10minSeek+15avgSeek-5maxSeek)/(3\sqrt{Cyl})$

$b=(7minSeek-15avgSeek+8maxSeek)/3Cyl)$

$c=minSeek$

其中 *minSeek*, *avgSeek*, *maxSeek* and *Cyl* 分别是 head switch, average seek, full stroke seek 和设备的最大柱面数.

表 4- 1 模拟试验采用的磁盘参数

Disk model	Quantum Atlas 10K Ⅱ 73.4GB
Cylinders	17337
Total tracks	346, 740
Sector per track	301- 528
Bytes per sector	512Bytes
Head Switch	0.6ms
Average seeking time	4.7ms
Full stroke seek	12ms
Revolution	10000rpm
Media transfer	31.0- 42.0MB/s
Year	2000

5.2 负载合成

模拟中, 请求产生器按照 iostone 和 postmark 两种 benchmark 分布合成负载产生 I/O 请求, 进行测试.

Iostone benchmark 的负载合成方式在引言中有过论述. 但是因为模拟中盘阵的逻辑单元大小为 1K 字节, 而 iostone 源代码的最小文件大小为 256 字节, 所以模拟中我们将源代码中的文件大小扩大 4 倍, 其余的如文件比例保持不变. 因此在此类测试中, 文件大小有 1KB、2KB、4KB、8KB、16KB、32KB、64KB、128KB 和 256KB, 相应大小的文件比例为 32.3%、16.2%、16.2%、16.2%、8.1%、8.1%、2.0%、0.5% 和 0.5%. 各文件访问

的频率是相同的.

对于 postmark benchmark, 合成负载时, 我们模拟的事务中, 只有创建和读两种文件操作, 也就是说每个事务是读一个文件后, 再写一个文件, 读写的文件号是随机的, 长度为该文件的初始创建的文件大小, 而初始创建的文件大小为 1KB 到 100KB 之间均匀分布(类似于 email 系统的负载, 读一封邮件, 然后回复一个), 共有 1000 个文件.

请求产生器有 10 个并发的线程, 根据 iostone[16] 和 postmark[12] 两种 benchmark 源码规定的方式, 产生一个读写文件的 I/O 请求. 每个线程产生并提交请求后, 等待执行结果然后发出后续的请求. 模拟中, 缓冲区大小不受限, 一个请求一次读写整个文件, 而不是象源代码中一个请求一次只读写一个缓冲区.

5.3 试验结果

在试验中, 初始数据块分布有两种: round-robin 和 file-unit. 所谓 round-robin 是存储文件时, 首先在盘阵中随机选取一个设备存放第一块, 然后以 round-robin 方式将文件后续块存放在盘阵设备上; 而 file-unit 方式是指存放文件时, 随机选取盘阵中一个设备, 将文件的所有块都存放在这个设备上, 但是操作文件时, 必须按照数据块数目进行多次 I/O 请求才能完成, 而不是一个大 I/O 请求(该分布是为了模拟大 I/O 请求不能并发的情况). 优化策略的触发时间是盘阵每执行完 200 个请求, 进行一次优化调整.

测试比较中, 我们分两种 benchmark 的负载在不同的初始分布和不同的分布优化策略下性能比较, 其中分布策略包括初始分布、平衡超图划分、自适应负载平衡策略(磁盘冷却[13]) 和请求并发策略(超图划分[9]). 为了作图方便, 在图中我们规定 th1 为初始分布的请求平均响应时间, th2、th3 和 th4 分别为仅仅进行请求并发的优化策略(超图划分算法[9])、仅仅进行负载平衡的优化策略(磁盘冷却算法[13]) 和请求并发与负载平衡同时考虑(基于平衡超图的优化策略) 的请求平均响应时间. 模拟试验比较 4 者中盘阵的请求平均响应时间.

根据 iostone 产生负载测试系统的性能如图 1 和图 2.

图 1 是初始分布 th1 为 round-robin 方式. 其中: ①基于超图划分的自适应请求并发策略对系统性能没有优化效果[9], 而负载平衡策略有一定优化效果. 我们认为这是因为文献[9] 中基于超图划分只能对请求并发进行优化, 而 round-robin 的初始分布, 每个文件的所有块尽可能地分散在不同设备上, 而请求是操作整个文件, 所以保证了 iostone 的 I/O 请求的并发性已经达到了最大; 在这种情况下, 各设备上分布的文件数目基本相同, 但是由于文件长度不同, 使得盘阵的各设备负载不平衡(如果测试时间段选取的小), 因此负载平衡成了主要问题, 从而自适应负载平衡策略能带来优化效果. ②尽管在这种分布和请求模式下负载平衡是主要的, 负载平衡策略也有一定效果, 但基于平衡超图划分的策略在负载平衡的基础上, 响应时间减少了大约 10%. 这是因为如果仅仅只进行负载平衡[13], 数据重构时就会忽略请求的并发性(对大文件来说), 所以不能对盘阵进行全方位的优化; 而本研究提出的平衡超图划分方法则在负载平衡的同时, 又保证了请求并发.

图 2 的初始分布 th1 为 file-unit 分布. 图 2 中, 同初始分布相比, 负载平衡策略[13] 带来了一定优化效果(平均请求响应时间减少了 4.1%), 而请求并发[9] 带来的效果较为明显(平均请求响应时间减少了 23.3%). 这是因为在 file-unit 分布中, 请求并发是主要的. 在这些基础上, 本研究提出的平衡超图划分方法还能有改善(同初始分布相比平均请求响应时间减少了 33.2%).

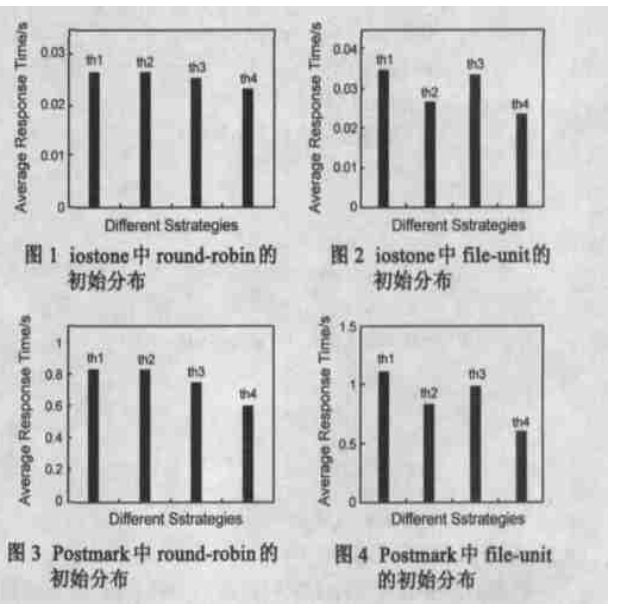

图 3 与图 4 是对 Postmark 的测试. Postmark 主要是测试事务处理速度. 在这种负载中, 我们只采用了文件的读取和创建作为事务的操作对. 模拟结果与对 iostone benchmark 的测试类似.

图 3 的初始分布 th1 为 round-robin 方式. 这种情况与图 1 中类似, 这种请求特点中, 负载平衡是一种主要因素, 因此, 负载平衡能带来优化效果, 而单纯的请求并发不能带来优化. 图 4 则不同, 初始分布 th1 为 file-unit 分布. 这种请求模式下请求并发是主要因素, 从而请求并发能取得更大优化效果. 而这几种情况中, 基于平衡超图划分的方法都是最好的数据分布优化策略.

模拟试验表明, 针对不同的数据分布, 基于平衡超图划分的方法能针对更多的模式发挥优化效果. 但是限于 benchmark 的请求负载模式, 该方法起到的优化效果还不是特别明显. 我们认为, 不同系统, 特别是数据库系统中, 由于请求模式和数据分布方式多种多样, 同传统单纯负载平衡或请求并发相比, 平衡超图划分策略会起到更好效果.

6 结论

盘阵中, 请求并发能减少请求的服务时间, 而负载平衡能使盘阵各设备充分工作. 原来的研究对二者是分开考虑的. 将二者的结合考虑, 势必能进一步优化性能. 本文结合二者进行研究, 并对已有的相关研究进行完善和改进, 对盘阵优化提出了基于平衡超图划分的自适应数据分布策略. 针对两种 benchmark 进行了模拟试验表明, 我们的算法比单纯只进行负

载平衡或请求并发性能高,系统的请求平均响应时间少.

数据库系统中,各种访问模式都有,且这些请求的特点是一些请求访问相同的数据块,且此类请求的频率特别高[9],在盘阵上建立数据库时,更需要对这些请求进行优化.因此下一步工作需要进一步分析多种系统的 I/O 特性,不断完善平衡超图划分策略,并将其应用到具体的系统 I/O 优化中.

当今基于超图的研究还不完善.我们的算法已经能适应很多访问模式的优化,但是可以肯定还有一些情况,这些算法是无效的.这些还有待进一步研究.

参考文献:

[1] E Anderson, et al. Hippodrome: running around storage administration [A]. Conference on File and Storage Technologies (FAST) [C]. Monterey, CA, 2002. 175- 188.

[2] E Anderson, et al. An experimental study of data migration algorithms [A]. 5th Workshop on Algorithm Engineering[C]. Denmark, 2001. 145 - 158.

[3] Copeland, et al. Data placement in bubba[A]. Proc. ACM International Conference on Management of Data [C]. Chicago, Illinois, United States, 1988. 99- 10.

[4] Daniel Ellard, et al. The utility of file names[R]. Harvard University Division of Engineering and Applied Sciences, 2003. TR- 05- 03.

[5] EMC Corp. EMC control center product description guide[M]. USA: EMC, No. 01748-9103, 2000. 2- 19.

[6] H Hermann, et al. An I/O architecture for microkernel-based operationg systems[R]. TU Dresden, Dresden, Germany, 2003: TUD-F103-08-Juli-2003.

[7] Hewlett-Packard Company. HP SureStore E auto lun XP user's guide [M]. USA: HP, No. B9340-90900, 2000. 10- 38.

[8] J Katcher. Postmark: a new file system benchmark[R]. Network Applrances Corp, USA: 1997. TR- 3022.

[9] M Koyuturk. Hypergraph based declustering for multi-disk databases [D]. M S Thesis, Bilkent University, Computer Engineering Dept, 2000.

[10] Edward K Lee, Randy H Katz. An analytic performance model of disk arrays[A]. Proc. of ACM SIGMETRICS Conf. on Measurement and Modeling of Computer Systems [C]. Santa Clara, California, USA, 1993. 98- 109.

[11] C Lu, et al. Aqueduct: online data migration with performance guarantees [A]. Conference on File and Storage Technologies (FAST) [C]. Monterey, CA, 2002. 219- 230.

[12] J Katcher. Source code of Postmark. http://www.netapp.com/ftp/postmark.c[EB/OL], 2004.

[13] P Scheuermann, et al. Data partitioning and load balancing in parallel disk systems[J]. VLDB Journal, 1998, 7(1): 48- 66.

[14] S Shekhar, D R Liu. Partitioning similarity graphs: A framework for Declustering problems[J]. Information Systems, 1996, 21(6): 475- 496.

[15] Xiaohui Shen, Alok Choudhary. DFS: a distributed parallel file system [A]. International Conference on Parallel Processing (ICPP'01) [C]. Valencia, Spain, 2001: 1- 15.

[16] iostone benchmark. http://www.cs.earlham.edu/charliep/~benchrmarks/iobench/iostone.c.original[EB/OL], 2002.

[17] D Roselli, et al. A comparison of file system workloads[A]. 2000 USENIX Technical Conference[C]. UC Berkerley, 2000. 41- 45.

作者简介:

刘　军　男,1971 年生,博士,主要研究方向并行 IO 和高速信号传输等. E-mail: muplj@tom.com

杨学军　男,1963 年生,教授,博士生导师,主要研究方向高性能计算等.

王俊伟　男,1976 出生,博士研究生,研究方向并行计算.

唐玉华　女,1962 年生,教授,硕士生导师,主要研究方向计算机网络.