

一种高性能的磁盘 I/O 系统¹

靳超 周枫 郑纬民

(清华大学计算机系, 北京 100084)

摘要: I/O 是当前计算机系统的主要性能瓶颈, 这是因为作为主要外部存储设备的磁盘由于其机械运动的本质特征导致其性能无法与内存的性能匹配。很多研究侧重于如何提高 I/O 的带宽, 而忽略了一个棘手的问题: 如何降低磁盘操作的延迟。这个问题对于同步磁盘操作尤为重要, 而实际中小量数据的同步操作在磁盘 I/O 操作中占有很大比例。本文介绍了一种通过牺牲磁盘的存储空间来降低 I/O 延迟的方法, 将该方法应用于解决小量数据的同步写操作问题能够取得较好的效果。同时该方法能够确保数据的持久性不受影响。

关键词: 磁盘缓存, 日志, 低延迟, 持久性

Trading Capacity for Performance in Disk System

JIN Chao, ZHOU Feng, ZHENG Wei-min

(Department of Computer Science and Technology, Tsinghua University, Beijing 100084, China)

Abstract: Now I/O is the bottleneck of performance in computer system. Although disk is the main second storage device, there is a performance gap between disk and RAM because of mechanism characters of disk system. And reducing the latency of synchronous operation is a very difficult problem. At the same time, synchronous operation of small data is general in I/O operation. Most research focus on how to improve the bandwidth of I/O, this paper recommend a method trading capacity for reducing latency of I/O. And this method can insure data durability.

Keyword: Disk Cache, Log, Low Latency, Durability

1. 引言

当今计算机系统的主要性能瓶颈在于 I/O 环节。这主要是由于 I/O 系统性能的提高远远低于 CPU 和内存性能的提高。根据 Moore 定律, CPU 性能和内存容量每 18 个月就可翻一番。但是作为外存的主要设备——硬磁盘, 由于机械运动的本质特征导致性能的提高非常有限, 每年约 7%^[1]。这样可由 Amadl 定律得出, 如今 I/O 的性能在系统整体性能中占有举足轻重的地位。不过需要指出的是磁盘存储密度每年提高约 60%, 这一点甚至高于 CPU 和内存的提高速度。实际上真正限制磁盘性能的主要因素是延迟, 每年只能增长约 10%^[2]。目前优化磁盘性能的各种技术主要侧重于如何提高数据传输的带宽, 很少有侧重于降低延迟的。

从软件角度讲, 可以采用多种延迟隐藏技术来降低 I/O 访问对系统性能的影响, 比如预取技术和写缓存技术等。但是在很多情况下这种方法是不起作用的, 尤其是需要同步 I/O 操作时。根据^[3], 通用文件系统中, 写操作占大多数, 约 57%, 同步操作比例占 50%到 75%, 而且写操作中对元数据的访问占 67%到 78% (而一般出于维护文件系统一致性的考虑, 对元数据的写访问需要通过同步操作完成)。从这些数据中可以得出 I/O 的同步写操作是性能瓶颈的主要原因。之所以如此, 主要是由于文件系统为了保证数据的完整性必须通过同步方式来完成对元数据以及重要用户数据的写操作。另外对于数据库系统而言, 同步操作几乎决定了其性能。

本文将介绍一种高性能的日志磁盘设备, 该设备是一种将磁盘写操作的性能优化到接近理想程度的日志设备。利用该设备可以在很大程度上提高小量数据同步 I/O 操作的响应速度, 从而改善 I/O 延迟对系统整体性能的影响。

2. 原理

2.1 思路

首先我们从产生小量数据 I/O 延迟的原因入手。以 IBM UltraStar DDYS-T3695 SCSI 磁盘²为例, 全程寻道时间约为 12ms, 也就是平均寻道延迟约为 6ms。盘片旋转一圈需时约 6ms, 即平均旋转延迟 3ms。数据传输速率在 20M/s 到 43M/s 之间。而根据磁盘的物理特性可知

¹本课题受 973 国家重点基础研究发展规划项目资助, 项目编号 G1999032702.

²本文中所采用的有关磁盘性能的数据均以此硬盘为依据

访问磁盘数据时间 = 控制器开销 + 寻道延迟 + 旋转延迟 + 数据传输时间。

其中控制器开销基本恒定，一般约为 1 至 2ms。

从磁盘访问时间公式中可以得出只有顺序访问磁盘才能尽量减少寻道延迟和旋转延迟在总时间所占比例。而对于小量数据和大量数据分别而言，磁盘所能提供的有效带宽差别很大。以 4k 数据和 1M 数据为例，在一般情况下 4k 数据的访问时间为 11.2ms 左右，有效的数据传输带宽为 0.36M/s。而访问 1M 数据需时约 61ms，有效数据传输带宽为 16.4M/s。所以小文件的传输速率非常低。

通过研究以往提出的各种磁头调度算法，我们认为想要使得磁盘的写操作性能和读操作性能同时达到最大化程度很难做到。因此为了解决小量数据的同步写操作问题，我们设计了一种高性能的日志磁盘设备。在计算机领域中，日志是很常用的容错技术，尤其在数据库系统中以及分布式系统均采用了这种技术。一般而言，日志使用追加的方式写入数据。而且很多环境中，只在必要时（比如系统崩溃后）才读取日志中的数据，一般而言日志只是作为一个备份。作者设计的这个设备首先符合日志的要求，向日志中写入数据是通过追加的方式完成的。同时由于日志一般主要提供写操作的访问，所以本日志设备也主要考虑对写操作的性能提供最大程度上的优化。

基本的想法是通过牺牲部分存储空间来换取高性能的写操作。要想把写操作的延迟降低到最低的程度只有一个方法，就是在临近磁头的位置执行写操作命令。这种做法能够基本去除寻道延迟和旋转延迟的开销，但是必须注意的是使用这种方法写入新数据时不能覆盖原来的有效数据。因此，我们不将磁盘的所有空间都用来保存有效的用户数据，而是使某些空间空闲，从而使得有效数据之间的距离间隔足够大。然后就可以在确保新写入的数据不会覆盖原来有效数据的情况下，基本达到在临近磁头的位置执行写操作命令的目的。

具体做法是，根据当前负载的情况确定应该在磁盘的哪个位置上执行写操作命令。当负载较少时，因为这个日志磁盘是专用盘，所以我们尽可能的保证日志磁盘中的磁头总是等待在某个空闲的磁道上，因此在一定程度上能够基本消除寻道延迟带来的开销。然后一个问题就是如何消除旋转延迟。若要磁头不需要等待盘片的旋转而直接将数据写入当前所在的扇区上，则必须能够知道当前时刻盘片旋转到了哪个位置上。也就是说，磁头位于磁盘的哪个扇区上就向该扇区中写入数据。这样才能将旋转延迟的开销降低到接近理想的程度。然而，目前磁盘所提供的接口中，没有这项功能。因此，我们独立研制了一种能够预测出磁头当前位置的算法。根据该算法能够得到某个时刻磁头相对于扇区的物理位置。于是向磁盘发送写操作命令时，将预测出的位置作为写命令中的位置参数，就能在很大程度上降低旋转延迟的开销。当系统负载较重时，直接采取以上做法可能导致系统需要等待磁头在磁道间移动，因此需要采取一种调度策略将整体的延迟降低到最小程度。下面首先介绍估测磁头位置的算法，然后详述负载较重时的调度策略。

2.2 估测磁头位置

算法描述：假定，在初始时刻 t_0 ，磁头位于位置 s_0 ，那么根据算法能够预测出 t_1 时刻磁头的位置 $s_1 = f(s_0, t_1 - t_0)$ ，其中 $t_1 > t_0$ 。

预测磁头位置是根据扇区的物理位置进行的。磁盘的物理地址 $C:H:S$ 中，需要预测的仅仅是扇区 (Sector) 位置。假定初始时刻 T_0 磁头位于 $C_0:H_0:S_0$ 扇区位置，那么预测 T_1 时刻磁头所在扇区位置 $C_1:H_1:S_1$ 的公式如下：

$$S_1 = \left(\frac{(T_1 - T_0) \bmod RT}{RT} * SPT + S_0 \right) \bmod SPT \quad (1)$$

其中 SPT(Sectors per Track)表示当前磁道上的扇区数目，RT(Rotation Time)是盘片旋转的周期， S_0 和 S_1 均为同一磁道上扇区的物理地址， $T_1 > T_0$ 。因为只有盘片在不断旋转，而磁柱和磁头是可以通命令来控制的。所以磁柱和磁道的位置不需要预测。需要注意的是当 $C_0:H_0 \neq C_1:H_1$ 时，需要将 S_0 转换为在磁道 $C_1:H_1$ 上相应的值。这一点很好实现，因为盘片上所有的磁道都是同心圆，所以各个扇区之间的相对位置是固定不变的。

具体的实现公式如下：

$$S_0' = S_0 * (SPT' / SPT) \quad (2)$$

其中 S_0' 为 S_0 在磁道 $C_1:H_1$ 上对应的物理扇区位置， SPT 为磁道 $C_0:H_0$ 上的扇区数目，而 SPT' 为磁道 $C_1:H_1$ 上扇区的数目。

然而实际应用时，公式(1)还有缺陷。因为从处理器向驱动设备发出命令到该命令在磁盘驱动器上被执行是需要一段时间的，所以需要修改公式(1)。另外盘片的旋转本身是有误差的，必须保证误差不能影响公式的准确率。将公式(1)修正为公式(3)

$$S1 = \left(\frac{(T1 - T0 + Adjust) \bmod RT}{RT} * SPT + S0 \right) \bmod SPT \quad (3)$$

其中 Adjust 是修正参数，需要通过测试得出。实验中使用的值为 1000ms。

本公式成立的一个重要条件就是磁盘旋转是否稳定。实际上，当前所使用的温彻斯特硬盘在室温化境下稳定性非常高，其旋转周期的误差小于 1%。所以能够满足算法要求。

另外一点需要指出的是，本算法是基于物理地址进行预测的。而实际硬盘提供给文件系统的接口使用的是逻辑地址。而逻辑地址与物理地址间的相互转换是隐藏在硬盘内部的。因此必须通过软件的方法将这些参数提取出来。并且还需要指出的一点是预测算法需要知道磁盘旋转周期的精确数值。而一般硬盘厂家所提供的手册上给出的磁盘旋转周期都不准确。所以也需要通过软件的方法将这个数值测试出来。关于这些问题的解决，由于篇幅所限作者会另写文章叙述。

2.3 优化策略

如果在每个磁道上只做一次日志记录，当本磁道的记录完成后立刻将磁头移动到下一个磁道上，对于这种做法在负载不密集的情况下可以保证每次记录的等待时间不超过 1.2ms。但是如果负载比较密集，则需要附加移动磁头所需要的时间。磁头移动一道所需要的时间从 0.5ms 到 0.9ms 不等，因此这一部分延迟实际上也是比较大的。另外，目前磁盘的密度相当大，单个磁道的总容量也很大，约为 100 到 200K 左右。而一般一条日志的长度约为几 K 到几十 K 不等，因此如果每个磁道仅记录一条日志，浪费的空间是很大的。在这种情况下，我们提出另一种策略：可以在同一个磁道上记录多个日志。这样对于一个已经有了记录数据的磁道而言，再次发送记录数据必须保证不会覆盖原有的有效数据，所以可能需要等待一段时间。如果需要等待的时间小于磁头移动一个磁道所需的时间，那么就在同一个磁道上记录日志；如果需要等待的时间长于磁头移动一个磁道所需的时间，那么就将磁头移动到下一个磁道上。

下面仅以一个磁道上记录两条日志为例讨论一下优化策略的可行性。如果要在同一磁道上记录多个日志的话，必须判断为了保证不覆盖有效数据所必须等待的旋转延迟是否小于磁头移动到下一磁道的延迟。决策方案如下。首先计算等待延迟 WT。

$$WT = \begin{cases} 0 & \text{if } (S_1 - S_0) \geq Record_Length \\ \frac{\|S_1 - S_0\|}{SPT} * RT & \text{if } (S_1 - S_0) < Record_Length \end{cases}$$

其中 S_1 为设备接收到写数据命令时磁头在磁道上的相应位置， S_0 为上次记录起始的位置， S_0' 为上次记录结束的位置，Record_Length 为本次记录的长度。如果 $WT <$ 移动磁道所需时间，则继续在当前磁道上记录数据；否则将磁头移向下一个空闲磁道。如图 1 所示，每个圆均代表一个磁道，实线代表日志，假定磁盘是顺时针旋转的。 S_0 到 S_0' 的实线代表上次的日志记录。点 A 到 S_0 的长度为 Record_Length。如果 S_1 位于从 A 顺时针到 S_0' 的区间内，则从这个位置开始写记录数据不会覆盖已有的有效数据。相反，如果 S_1 位于从 A 逆时针到 S_0' 的区间内，则必须等待 WT 的时间将有效数据避开。

这种策略在负载繁忙的状态下，可以在一定程度上提高系统响应时间。同一磁道上记录的数据越多，则寻道带来的延迟就越小；但是同时覆盖有效数据的可能性就越大。因此首先需要在理论上估算一下单个磁道上最适宜记录几条日志。

假定磁道上有 S 个扇区，每个记录长度为 L 个扇区。

如果每个磁道上保存两个日志记录，则写的二次记录遇到有效数据的可能性为

$\frac{1}{S} \sum_{i=1}^L i * t$ ，也就

是 $\frac{L}{S} * \frac{(1+L)}{2} * t$ (其中 t 为等待一个扇区需要的延迟)。只要保证 $\frac{L}{S} * \frac{(1+L)}{2} * t <$ 移动一道磁道

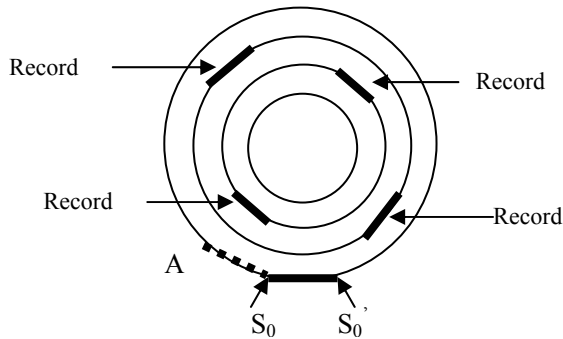


Figure 1 Optimized Record on Cache Disk
图 1 日志设备优化记录示意图

所需时间，就能保证每个磁道上记录多个数据的延迟小于移动磁道所需的时间。假如 $S=400$ ， $RT=6000us$ ，则 $t=(1/400)*RT$ ，移动一道磁道所需的时间为 $500us$ 。则通过上述公式可粗略估算出 L 约为 163。

如果每个磁道上保存三个日志记录，则写的三次记录遇到有效数据的可能性为 $2L/S$ 。而平均的旋转延迟为 $\frac{1}{S} \sum_{i=1}^{2L} i * t$ ，也就是 $\frac{2L}{S} * \frac{(1+2L)}{2} * t$ （其中 t 为等待一个扇区需要的延迟）。只要保证 $\frac{2L}{S} * \frac{(1+2L)}{2} * t < \text{移动一道磁道所需时间}$ 就能保证每个磁道上记录多个数据的延迟小于移动磁道所需的时间。条件同上，则可粗略估算出 L 约为 83。

而一般而言同步操作记录的数据在几 k 左右，也就是记录长度 L 很难超过 20。所以，在一个磁道上记录多条数据是可行的。但是在实际的实现中，由于记录多条数据需要附加的开销保证不会覆盖有效的数据，因此每个磁道上记录的数据不宜过多。

2.4 从日志磁盘中读取数据

正常使用情况下，读取日志磁盘的接口是通过记录的 ID 进行的。系统内部维护日志记录的信息表，能够通过记录 ID 找到该日志的地址与长度。然而在日志工作时进行大量读取会降低写记录的效率。由于日志使用环境的特点，所以在日志适用的环境下从日志盘中读取数据的情况很少发生。

3. 利用高性能日志设备提高磁盘的同步操作

由于该日志设备能够将磁盘写操作的性能发挥到最大极限，所以可以考虑利用该设备提高同步写操作。上文中已经说过，之所以需要同步写操作是为了保证数据不丢失。我们将内存中的缓存与高性能的日志设备相结合设计了一种二级缓存结构。系统结构如图 2 所示。

系统中部分 RAM 缓冲区和日志缓存盘结合在一起形成虚拟的可靠内存（Virtual Reliable Memory, VRM）。其容量相当于内存的容量。具体 VRM 中使用的是哪部分内存由缓存管理算法决定。内存中那些在缓存盘上有备份的数据被称为处于 log 状态。VRM 使用的就是这些处于 log 状态的那部分内存。从文件系统层看来，向 VRM 中写入数据的性能要低于内存但是高于通常的磁盘操作。而从 VRM 中读出数据与从内存中读出数据没有区别。

3.1 读操作

需要从数据盘上读取数据时，不经过日志磁盘，数据直接从数据盘读出复制到 RAM 缓冲区中。其优点是本系统不妨碍文件系统对读操作所做的任何优化。由于本系统能够减少数据盘 I/O 写操作的整体响应时间，所以读操作的性能应该也有所提高。

3.2 写操作

写操作分成两种情况：同步操作和异步操作。对于异步数据，同样不经过日志缓存盘，直接从内存缓冲池到达数据盘中。而同步数据首先要写到缓存盘上，然后向进程返回完成信号。此时数据仍就保留在内存中，只是转换成为异步数据。然后通过异步写操作将数据写入数据盘中。而异步操作由后台进程完成，不影响应用程序的性能。

3.3 系统崩溃后恢复数据

系统崩溃后，如果缓存盘中留有未能转入到数据盘的数据则需要从中将这些数据读出，然后转入到数据盘中。虽然内存中的数据丢失了，但是可以从缓存盘上恢复过来，这一点充分保证了数据的持久性。本过程不妨碍文件系统一致性检查。但是这个过程需要早于文件系统一致性的检查工作完成。

与一般文件系统及数据库系统中从日志恢复数据的不同之处在于，首先需要从日志盘上的杂乱无章的数据找到有效的记录数据，然后在从这些有效的记录数据中将需要恢复的数据转移到数据盘中。

为了恢复缓存在日志盘上的数据，需要在给这些数据加上标志。每一次写入缓存盘的数据我们称为一个记录。每个记录由一个头部和记录内容构成。记录内容就是需要缓存的数据。而每个记录头部中的主要字段如下表所示：

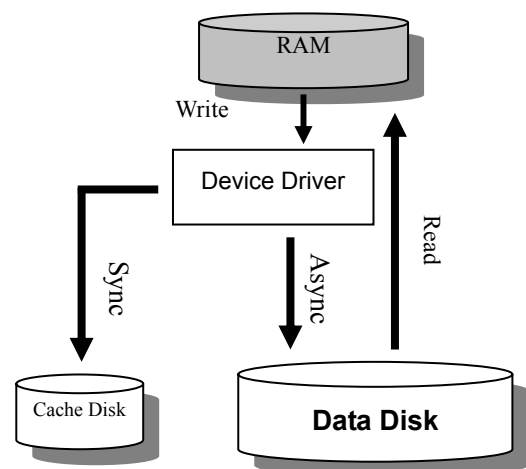


Figure 2 Architecture of Log Device
图 2 利用日志设备提高磁盘同步操作的结构示意图

Magic	Epoch	Sequence	last_checkpoint_sequence
-------	-------	----------	--------------------------

表 1 记录头部中的主要字段

下面讲述对其中的主要标志信息的含义。Magic 是记录起始的标志，本系统中采用 0xfe 作为起始标志。如果某个扇区的第一个字节与 0xfe 相等，则认为这个扇区就是一条记录的起始位置。Epoch 和 sequence 共同确定各个记录间的时间顺序，通过这两个域在各条记录间定义了一种偏序关系。当系统崩溃后，就按照这种偏序关系在缓存盘上查找未被写入数据盘的记录。Last_checkpoint_sequence 是最后一个做检查点的记录在当前这一批日志中的序号。通过这个标志位确定未被写入硬盘的起始记录的地址。如果日志的数据中，有某个扇区的第一个字节与 magic 相等，则必须将之替换为其他的任意数据（本系统中替换为 0）。同时必须在记录头部加以标识。

恢复数据时，需要搜索整个日志盘的所有空间将有效数据提取出来。寻找最后一个记录的过程需要将整个磁道上的内容读入内存，确定其中哪些数据是日志，然后在这所有的日志中找到日期最近的那个。假定日志空间为 10000 个磁道，读取每个磁道时间约为 $(252k)/(40M/s)=6.3ms$ 。由于整个缓冲磁盘可以看作是循环的日志空间，所以可以使用折半查找的方法确定最小的那个日志记录。这样折半查找的复杂度为 $\log_2^{10000}=13.3$ ，也就是确定最小的日志记录所需的时间约为 100ms。即便是再附加一些其他的处理开销整个时间应该不会超过 1 分钟。接下来就是要把缓存数据转入的数据盘中了。在作者实现的原型系统中，未写入到数据盘中的数据不会超过 500 个，这是由检查点机制决定的。所以整个恢复过程所需的时间不超过 5 分钟。

4. 性能测试结果

我们对日志盘的写操作以及利用日志盘提高 I/O 系统的同步操作进行了性能测试。测试环境如下：CPU 为 Pentium III 700，日志磁盘与数据磁盘均为 IBM UltraStar DDYS-T3695 SCSI，操作系统为 Linux (Kernel 2.4.5)。

4.1 估测磁头位置算法的准确度

如下表所示，估测磁头位置的算法准确度还是比较高的。表中所示的理想情况下的数值是根据硬盘所提供的参数计算出来的，作为参考值。从表中可以看出，利用本算法的 I/O 带宽接近理想值的一半。这主要是由于实际系统中的某些命令开销以及为了降低误差对算法的影响而造成的。

数据块长度 (KB)	1	2	4	8	16	32
响应时间 (us)	836	908	957	1059	1241	1776
带宽 (M/s)	1.196	2.203	4.18	7.554	12.893	18.018
理想的响应时间 (us)	425	450	500	600	700	900
理想的带宽 (M/s)	2.35	4.44	8.00	13.33	22.86	35.56

表 2 磁头预测算法的性能比较

4.2 日志盘所能提供的写操作性能

测试结果如图 3 所示。图中纵轴单位为秒，横轴单位为 KB。测试是在负载密集情况下得到的。所示分别为每道上日志一条、两条和三条记录的性能结果。由图中可以看出同一磁道上日志的记录数目越多，其性能就越高。当然，日志记录数目增加到一定程度后，负面影响会逐渐增大。所以实际实现的过程中应该选择一个最佳的记录个数保证性能达到最佳程度。而这个数目的选择与每条记录的长度以及磁道的长度均有关系。在我们的实际测试过程中，认为每道记录三条比较合适。从图 3 中可以看出，日志三条记录同日志一条记录相比性能提高约为 20%左右。

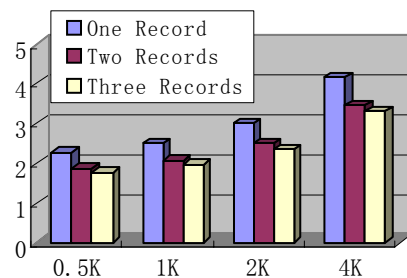


Figure 3 Performance of Log Device

图 3 日志盘性能测试结果比较

4.3 利用高性能的日志设备提高 I/O 子系统的同步操作

本节测试了利用高性能日志磁盘作为磁盘缓存提高 I/O 同步操作情况下的性能结果，如图 4 所示。使用磁盘缓存与不使用磁盘缓存的情况下 I/O 子系统同步操作的性能差距非常大，性能提高约为 4 倍左右。图中纵轴单位为秒；横轴单位为 KB。

5. 结论

通过以上测试结果可以看出,日志缓存磁盘系统本身的确能够提供高性能的磁盘同步写操作,同时能够确保数据的持久性。利用该日志设备可以在很大程度上提高 I/O 子系统的同步写操作性能,而且不妨碍文件系统的完整性以及对读操作进行的各种优化设计。

5.1 与相关系统比较

Soft-Update[4]是密歇根大学提出的一种提高同步 I/O 操作的方案。这种方法利用写缓存的方式,将写操作尽量保留在内存中,然后通过异步方式写入磁盘。Soft-Update 方案通过维护各个数据块之间的依赖关系保证写入文件系统的数据块满足文件系统的一致性的要求。由于依赖于内存,这种方法的性能要比我们的系统高很多。但该系统无法保证数据的持久性,这正是我们相对于该系统的优势。

DCD [5]是罗特岛大学的 Qing Yang 等人提出的一种利用 NVRAM 和缓存盘提高小量数据写操作的方法。该系统从结构上除了需要日志磁盘作缓存外,还需要附加的 NVRAM 硬件支持来保证数据可靠性。同时该系统在回收日志空间时需要读取日志缓存盘的数据,然后转入数据盘中。虽然能够在写操作负载密集时提高系统的整体性能,但是却将一次写磁盘的操作增加为三次,这样对于系统性能的负面影响是难以衡量的。与之相比,本系统在达到相同的数据持久性支持条件下,不需要多余的硬件支持,而且对系统性能没有负面影响。在同等硬件条件下,当 I/O 繁忙时本系统的性能在保证同样数据可靠性的前提下能够比 DCD 系统高出一倍左右。

Trail[6]是 StoneBrooks 设计的一种以磁道为基本日志单位的缓存系统。该系统只有单一的日志策略,不能根据负载环境的不同采取相应的优化措施。与之相比,本系统对于负载环境有更好的适应性。能够针对不同的负载状况采取不同的策略。在同步写操作请求密集时,本系统具有更好的性能。性能提高比例可达 20% 以上。同时本系统对磁盘空间的利用率更高。

5.2 进一步工作

目前我们仅仅在单独的硬盘上尝试了利用磁头位置的信息优化写操的性能。进而可以考虑将这种功能融入到 RAID 系统中。因为 RAID 系统中,小量数据的写操作是性能的主要瓶颈。可以考虑利用本文所述的设备提高 RAID 系统中的小量数据写操作问题。

另一方面,同步操作在许多环境中非常重要。比如数据库系统的事务处理,Journaling File System 中的日志记录等。这些环境中也都是由于需要保证数据可靠性而必须采用同步操作,可以考虑将本系统利用在这些环境中在保证数据可靠性的前提下提高性能。

当前同步 I/O 延迟的问题主要是通过 RAM 来解决的,比如内存或磁盘系统上的 RAM。而在 I/O 繁忙的情况下, RAM 的容量必须足够并且要通过 NV-RAM 来保证数据可靠性。这样就必须付出高昂的代价。随着嵌入式系统的发展,将来的磁盘系统中的嵌入式 CPU 功能会比较强,因此可以考虑将本论文中所介绍的方法应用到将来的磁盘系统中。这样能够在最大程度上发挥本方法的优势。

6. 参考文献

- [1] C. Ruemmler and J. Wilkes. "An introduction to disk drive modeling." IEEE Computer, 1994, 27(3):17-29.
- [2] Growchowski. E. Emerging Trends in Data Storage on Magnetic Hard Disk Drives. In: Datatech, ICG Publishing, 1988, 11-16.
- [3] Chris Ruemmler, John Wilkes. UNIX disk access patterns. In: Technical Conference Proceedings of USENIX. Sa Diego, CA. 1993, 405-420.
- [4] M.K. McKusick and G.R. Ganger. Soft Updates: A Technique for Eliminating Most Synchronous Writes in the Fast File System. In: Proc. of the 1999 Usenix Technical Conference, Freenix Track, 1999, 1—17.
- [5] Yiming Hu, Qing Yang. DCD-Disk Caching Disk: A New Approach for Boosting I/O Performance. In: Annual International Symposium on Computer Architecture, 1996.
- [6] T. Chiueh. Trail: A Track-Based Logging Disk Architecture for Zero-Overhead Writes. In: Proceedings of 1993 IEEE International conference on Computer Design ICCD'93, Cambridge, 1993, 339-3443.

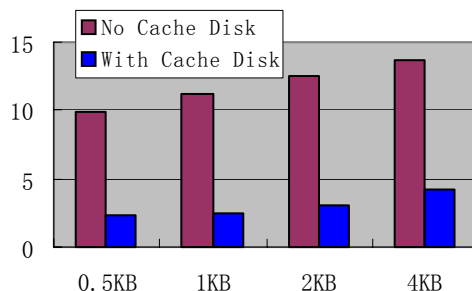


Figure 4 Effects of Log Device
图 4 磁盘缓存在同步写操作中的作用

作者简介:

靳超, 男, 99 年起攻读清华大学计算机科学与技术系博士学位, 专业方向为分布式计算、并行计算、分布式存储等。

周枫, 男, 2000 年起攻读清华大学计算机科学与技术系硕士学位, 专业方向为分布式计算、并行计算、分布式存储等。

郑纬民, 教授, 博导, 清华大学计算机科学与技术系高性能计算研究所所长。学科专业为计算机组织与系统结构, 并行处理与分布式系统, 在国内外著名学术期刊与会议上发表论文数十篇。

第一作者联系方式:

靳超 通信地址: 清华大学计算机系高性能计算研究所

邮编: 100084 办公电话: (010) 62785592

宿舍电话: (010)62779115 Mobile: 13911235727

Email: jinchao99@mails.tsinghua.edu.cn