Linux 下一种磁盘节能的预取算法①

姚 维 (湖南大学 软件学院 湖南 长沙 410082)

摘 要: 数据预取常用来提升系统的性能与吞吐量，对磁盘的能耗考虑甚少。针对此问题，在传统算法之上通过延迟磁盘的异步预取，合并磁盘 I/O 操作，减少磁盘的能耗状态切换，延长连续休眠时间来达到节能的目的。也通过基于真实运行状态的模拟,对预取算法进行了评估和验证，得出改进后的预取在不影响性能的前提下比标准预取节省 17%的能量。

关键词: 数据预取;访问模式;磁盘 I/O;空闲;能量有效

A Disk Energy Saving of Perfecting Algorithms in Linux

YAO Wei (Software School, Hunan University, Changsha 410082, China)

Abstract: Data prefeching is used to improve system performance and thuoughput, which ignores the disk energy comsumption. As for this problem, a delaying asynchronism prefeching, clusting disk I/O handle way is proposed based on tradition prefeching to reduce the disk state switching and extend the length of disk idle phases for saving power. We also validate and evaluate this algorithm based on the really running simulator. Our simulations show that compared to stand prefeching with negligible loss performance, our algorithm saves 17% more disk energy.

Keywords: data prefeching; access pattern; disk I/O; idle; energy efficiency

1 引言

随着现代计算机科学技术的发展，基于 Internet 和 web 技术构建的分布式服务模式日渐流行，政府、机关、企业单位等都全力打造自己的数据中心[1]，信息化时代给人类带来了空前的数据处理效率。但因为信息量的增大、业务系统的增强和用户服务的要求不断提高，在看似没有污染的数据中心背后有着巨大的能源消耗。据相关资料统计[2]，美国的数据中心在 2005 年消耗的电力占全美当年电力消耗的 1.2%，现在已经达到整个国家的 1.5%。如何降低数据中心的能耗，打造“绿色数据中心”成为业界关注的焦点。

磁盘是数据中心的主要能耗大户[3]，现代磁盘在能耗方面做了相关节能方面的考虑，但多基于硬件级的设计，这种依赖于设备级的能耗管理还达不到低能耗的目的。预取是一种软件级磁盘节能[4]的有效方法，它通过隐藏磁盘 I/O，延长磁盘连续休眠的时间来节能。但以往的多是一种平滑预取，一方面过于保守，另一方面不能均衡能耗与性能。本文就是针对此问题，在不影响系统性能的前提下，从能耗角度出发，延迟异步预取，合并磁盘操作集，自适应调整预取粒度，使之在工作时任务饱满，连续休眠时间足够长，从而达到节能的目的。本文最后还给出了模拟实验结果，说明该算法的有效性。

2 节能预取算法

2.1 磁盘能耗影响因素

现代磁盘支持多种能级服务模式，一般有工作(磁盘读/写)、休眠、空闲等能量状态。工作状态是一种高能耗状态，当应用程序请求服务时，系统将磁盘调度到工作状态，服务过程中磁头的不断移动，盘片的高速旋转，都需要大量的能量支撑。另外，当磁头移动到盘片指定区域读写数据时，磁头臂也需要进行移动，虽然过程很短，但也需要较大的电流。休眠状态没有磁头与盘片的物理运动，是一种能量有效的状态，

① 收稿时间:2009-10-21;收到修改稿时间:2009-11-24

但它的缺点就是给服务带来了延迟，一定程度上影响了性能。空闲状态是一种基于活动与休眠状态之间的过渡状态，它没有磁头的径向移动，但盘片仍然处于高速运转当中。该状态设计的好处就是当下一个请求到来时，磁盘能立刻反应，缩短了 I/O 等待时间。虽然该过程平衡了服务响应速度与能量消耗，但只适应于负载比较集中的情况，如果长时间没有服务请求，盘片仍处于空运转状态，这样也白白的浪费很多能量。通过状态[5]的切换为磁盘节能带来了机会，但它只对于有连续足够长的时间让磁盘停驻在低能耗状态节能才比较有效，如果对磁盘进行频繁的切换，这不但不能节省能量，反而降低系统的性能，而且容易造成磁盘故障。原因在于状态的切换本身也是一个高能耗的过程，从休眠调度到工作状态这个过程它的功率能达到读写时的 2-3 倍，时间也需要 2-3 秒[6]。因此如何通过预取，延长磁盘的连续休眠时间，减少磁盘能耗状态的切换成了磁盘节能的关键问题，也是要研究的重点。

2.2 预取的关键问题

磁盘有多种状态的切换，如果刚刚完成一次服务，把磁盘调度到休眠状态，现又有新的请求，磁盘需重新启动，直到服务完成。如果两次请求的时间比较短，磁盘还来不及休眠，又要回到服务状态，这个代价是相当高的。因此在什么时候将磁盘调度到休眠状态至关重要。假设磁盘支持三种状态，设$p_i(i=1,2,3)$分别表示工作、空闲、休眠时期的磁盘运作功率，$\mathrm{t}_i(i=1,2,3)$ 表示在一段时间 T 内各种状态维持的时间，$\mathrm{t}_{\text{spin-up}}$ 表示磁盘启动需要的时间，$E_{\text{spin-up}}$ 表示启动需要的能量，$\mathrm{t}_{\text{spin-down}}$ 表示停转需要的时间，$E_{\text{spin-down}}$ 表示停转需要的能量，$n_{\text{spin-up}}$ 表示时间 T 内磁盘启动切换的次数，$n_{\text{spin-down}}$ 表示时间 T 内磁盘停转切换的次数，则总个时间 T 内，磁盘消耗的总能量 $E(\mathrm{T})$ 可以表示为，

$$E(\mathrm{T})=\sum \mathrm{p_i}*t_i+n_{\text{spin-up}}*E_{\text{spin-up}}+n_{\text{spin-down}}*E_{\text{spin-down}}\ (i=1,2,3)$$

$\mathrm{T}=\sum \mathrm{t}_i+n_{spin-up}*t_{spin-up}+n_{spin-down}*t_{spin-down}$，由此公式看，磁盘能耗的大小与各种状态的持续时间，以及状态切换的次数紧密相连。当磁盘完成一次服务时，是继续保持在空闲状态还是调度到休眠状态，这与下次请求的时间关系紧密。从服务完成到下次请求 t 时刻到来之前，如果:

$$(E_{\text{spin-up}}+E_{\text{spin-down}}+(t-\mathrm{t}_{\text{spin-up}}-\mathrm{t}_{\text{spin-down}})*p_3)$$

小于 p_2*t，则表示切换到休眠状态能花费更少的能量。当两者相等时，能够解得一个固定的时间 t,我们称这个时间为能量均衡时间,只有当连续空闲时间大于均衡时间时，发生磁盘的休眠切换，才是能量最优的调度。

预取是将以后要用到的数据批量的放到内存缓存，当应用程序要用到该数据时直接从内存中获取，它有助于延长磁盘空闲时间，对磁盘的节能大大有益。但前提是能够准确的预取数据，它的正确率将直接影响系统的性能以及磁盘能耗的多少。准确的预取一方面能够估算下次预取的时间，另一方面也能调节下次预取的尺度。预取常见的有两种方案：启发性和知情的。对于知情的预取，系统提供一套预取 API，上层应用明确指示访问模式，调用适当的 API，完成预取工作。但它对应用程序不透明，具有一定的局限性。启发性的预取，对算法的要求比较高，它根据历史访问情况，自发的识别程序的访问模式进行预取决算，它对上层应用是透明的，也是一种应用比较广泛的预取。我们采取后者，利用历史的访问记录，识别程序的访问模式，自动调节预取数据的大小，聚集磁盘的读操作。

启发式的数据预取正确率依赖于访问模式识别率，预取支持的访问识别模式越多，总体识别率就越高。最常见的访问模式有随机、顺序等其它非典型模式。顺序访问模式占有比率比较高，也是最容易被识别出来的一种模式。访问模式一旦被识别之后，就可以处理预取最关键的问题：在什么时候预取，预取哪些数据，该预取多少？预取过早与过迟都有很大的负面影响，内存毕竟是一种有限的资源，过早可能可能会引起数据还没来得及访问就被内存管理回收，造成缓存抖动。另外可能过早的启动磁盘，打破了磁盘的休眠状态；或者限制了磁盘连续空闲的时间，致使磁盘根本就没有机会切换到低能耗状态。过迟预取又影响系统的性能。在什么时候最合适在一个预取周期中(这里我们假设本次预取完成到下一次预取开始的时间段称为一个周期)，在所有应用程序中，距离下一次磁盘访问最近的时刻之前刚好完成预取无疑是最理想的情况。一方面延长了磁盘的状态保持时间，另外一方面保证了性能。我们把这种预取叫主动预取，另一

种是被动预取，当预取数据失败，应用程序不得不从磁盘重新获得数据，它是一种同步预取，它有可能提前打破磁盘的能效状态，所以这种预取是需要尽量避免的。这里存在的另一个问题是对于单个应用程序而言，一次预取可能保证某个应用程序有相当长的一段时间没必要访问磁盘，但如果对多个应用程序而言，它们之间的磁盘访问没有很好的同步性，导致磁盘时刻要为不同的应用程序服务，这不仅不能达到节能的显著效果，而且不能充分利用磁盘的 I/O 带宽。所以要协调好总个应用程序 I/O 请求，对某些异步预取可以适当延迟，使应用程序之间能友好的同步访问磁盘。当新一轮的预取开始时，预取哪些数据？它一定程度上依赖于上次预取的数据位置，大小，访问模式以及应用程序真正的数据访问历史。一次预取既是本次预取的决策结果，也是下一次预取的依据。一旦历史的预取正确，就可以抛开保守的预取限制，自动的增加预取数据的大小，从上次预取后的位置开始，按照原来的访问模式向前或者向后的预取指定大小的数据，这个大小动态修改。基于这种模式识别的启发式预取算法不仅解决了预取的最关键问题，而且能自适应的调整预取的大小，是一种可行的方案。

3 算法的设计与实现

3.1 记录访问历史

借助访问历史来提高预取的精准度需要维护预取历史和访问历史两种状态量，预取历史主要记录每次预取位置、数据大小，访问历史主要记录访问数据的时间，位置。借助这两种历史我们就可以估计应用程序的访问模式以及预取的数据大小。我们用两个结构体维护这些历史，虽然完整的历史能更好的反映程序的访问特征，提高未来数据的预测率，但由于算法的时效性以及存储空间的开销，历史记录并非越多越好，所以我们只记录最近的几次预取历史。这些历史数据的获得需要实施的监控，为此我们在系统级增加监控模块，除了主要负责监控文件的访问请求之外，还负责实施监控系统可用的内存资源，磁盘 I/O 操作情况。有了这些数据就可以为预取状态结构 file_ra_stage 赋值，我们将简化系统传统的双窗口管理结构，仅仅保留预取的位置，大小，最大预取阀值，以及最近一次读的位置，并增加是否需要启动预取的标志。

3.2 启动预取时期

预取的时期还需要进一步确定，虽然上面我们讲到了一种最理想的情况，但这种理想时期很难把握。除了被动预取在任何时候毫无理由的需要启动之外，主动预取也需要时间量化。保守的预取是一旦监控到异步预取的内容被应用程序访问就立即启动新一轮的预取，它对延长磁盘连续空闲的时间支持不够，应用程序现访问的数据有可能是上上次预取的数据，甚至更远，它距离使用下一轮预取数据的时间还很长，本来可以让磁盘休眠更长的时间，却被这种平滑的预取破坏。因此在预取数据的过程中，我们在适当的预取页面设置一个预取标志，当该标志被访问到时，立即将预取状态中是否启动预取的标志设为真，当监控程序监测到该值为真时，按照预取历史的位置偏移启动新一轮的预取，并将该值设为假。这里我们将预取页面中的标志设在中间页开始位置。例如当共预取 32 页的数据时，我们将该标志放在 16 页开始的位置。

3.3 自适应调整预取粒度

现在来讨论预取尺度的设置与推进过程。在新打开的文件第一次预取时，预取历史中还没有初始化，则结构体中对该文件名的预取大小值为 0。由于第一次预取一般都是无条件被触发的，它的数据一定被访问到，我们做一次乐观的假设，假设该数据之后的相关数据在不久也将被请求，所以我们在请求的数据之后同时启动异步预取，并用默认的预取粒度初始化预取历史中的预取大小，设默认值为 4k。初次预取的数据有可能永远不会被访问，也有可能即将被访问。实时监控一旦发现初次预取的数据被访问到，则表示应用程序按照乐观假定的访问模式在进行数据访问，此时我们可以扩大预取的尺度，在原来的尺度上按倍增长，每一次的预取都在前一次的基础上扩大两倍，直到系统限定的一个阀值。Linux 系统中阀值常常设为 128K，即 32 页。为了不至于太保守，我们适当修改这个值，把它扩大一倍甚至更大。应用程序也不可能总是按照预取的数据进行访问，预取算法也没有办法知道应用程序当前访问流该在什么时候结束，所以总是有被动预取的情况发生。一旦发生被动预取，预示着上次预取的数据有可能是错误的数据，这个时候我们应该停止按照成倍增长的预取尺度，不能再借鉴上次预取的历史，此时将历史清空，以被动预取作为中心，重新初始化预取历史，仍按照原来的初始化方式进行，直到新一轮的预取失败，反复按照这样的方式进行推演。

3.4 内存管理

虽然实时监控能更多的获取可用内存资源，但不能保证每次预取都有足够的空间存放预取的数据，这时需要系统回收部分内存。这里仍采用经典的 LRU 算法首先回收预取缓存中从未过访问过，并且预取时间比较长远的数据。

我们在 linux 核为 2.6.22 的系统核上，修改了标准的预取例程，这里主要识别为顺序访问模式，随机模式不做异步预取，其简单流程如下：

```
Read()
{
    for(all page)
if(page not in cache )
        分配 page 加入 pagecache;
       call page_cache_sync_readahead()
else
If (启动预取标志为真)
call page_cache_async_readahead(); //预取操作
}
page_cache_async_readahead()
{
If (sequential )  //被识别为顺序读
Updata file_ra_stage object;
 Prefech data from disk and  add “prefech_flag” into middle prefeching  pagecache
}
```

4 模拟实验与结果比较

我们在环境为 Linux 2.6.20 ,2.0 GHz IntelCore 2 CPU,2G DDR2 内存的机器上进行了实验，磁盘采用 DiskSim 模拟器，将它配置成具有多种能级状态的磁盘，参照文献[6]的方法搭建实验环境并配置了如下模拟参数(见表 1)。

表 1　磁盘模拟能耗参数表

磁盘参数	参数值（单位）
磁盘容量	40GB
工作功率	2.1 W
休眠功率	0.2W
空闲功率	1.2W
启动(spin-up)能耗	6 J
停转(spin-down)能耗	3.2J
启动(spin-up)时间	2 秒
停转(spin-down)时间	1.5 秒

模拟器跟踪每次 I/O 操作的执行时间，请求数据的大小，以及各种状态维持的时间，通过这些数据就能计算出磁盘的一段时间 T 内消耗的能量 E(T)。

我们将修改了的预取算法加入到系统内核并重新编译安装。在配置好了的环境下，采用 Cello99 负载跟踪以及反复解压缩一个 1G 大小的文件作为数据测试请求，为了证明算法具有能量有效性，我们也在标准的系统内核上进行了实验，并采用相同的数据请求，执行相同的时间，得到了一组实验数据，如表 2。其中部分服务空闲时间长度结果如下图 1。

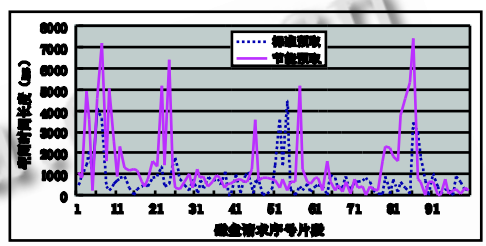

图 1　磁盘请求与空闲时间长度

从图 1 来看，节能预取的连续空闲时间得到了延长，与标准预取相比，较长时间的空闲次数也有所增多，并且更有机会超过均衡时间。

表 2　模拟实验结果数据

磁盘参数	标准预取	节能预取
服务次数	35079	28765
平均连续空闲时间	983.6 ms	1207.5ms
平均响应时间	346.7 ms	367.8ms

从表 2 的实验结果看，磁盘真正的服务 I/O 次数得到减少，其平均空闲时间也相应的延长，而且响应时间也没有明显的延迟。实验结果说明通过打破保守的预取，既保证了系统性能，磁盘的能耗得到了降低，按照模拟的参数计算，其能节能 17%。

5 结语

本文提出了一种通过动态调整预取粒度取，延迟磁盘异步预取，减少磁盘的能耗状态切换的能量有效预取算法，算法不仅能满足性能上的要求，而且为磁盘的低能耗状态的切换带来了更多的机会。

本文还通过基于实际运行状态的模拟,对该算法的重要参数和效果进行了探讨和验证,结果表明,在系统级引入合适的预取技术,对磁盘节能实际应用效果具有重要的意义。

(下接第 49 页)

(上接第 94 页)

参考文献

1 Manish M, Shivakant M. Enhanced Server Fault-Tolerance for Improved User Experience. International Conference on Dependable Systems & Networks (DSN'08), 2008,167－176.

2 http://server.ctocio.com.cn/news/175/7684175.shtml, 2008.

3 Pat B, Mootaz E, Tom K, et al. The Case for Power Management in Web Servers. Power Aware Computing, 2002.

4 Papathanasiou A E, Scott M. Energy Efficient Prefetching and Caching. Proc. of the Annual Conference on USENIX Annual Technical Conference, 2004, 22－35.

5 Fan X, Ellis CS, Lebeck AR. Memory Controller Policies for DRAM Power Management. International Symposium on Low Power Electronics and Design (ISLPED'01), 2001, 129－134.

6 Carrera EV, Pinheiro E, Bianchini R. Conserving disk energy in network servers. Proc. of the 17th International Conference on Supercomputing, June 2.