ISSN 1000-9825, CODEN RUXUEW
Journal of Software, Vol.21, No.8, August 2010, pp.1820−1833
doi: 10.3724/SP.J.1001.2010.03605
© by Institute of Software, the Chinese Academy of Sciences. All rights reserved.
E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

一种支持并发访问流的文件预取算法*

吴峰光+, 奚宏生, 徐陈锋

(中国科学技术大学 自动化系,安徽 合肥 230027)

File Prefetching Algorithm for Concurrent Streams

WU Feng-Guang+, XI Hong-Sheng, XU Chen-Feng

(Department of Automation, University of Science and Technology of China, Hefei 230027, China)

+ Corresponding author: E-mail: wfg@mail.ustc.edu.cn

Wu FG, Xi HS, Xu CF. File prefetching algorithm for concurrent streams. *Journal of Software*, 2010,21(8): 1820−1833. http://www.jos.org.cn/1000-9825/3605.htm

Abstract: This paper proposes and implements a demand prefetching algorithm, which uses relaxed sequentiality criteria as well as page and page cache states as reliable sources of information. It can discover and prefetch sequential streams mixed in random accesses. It can also support the interleaved access patterns created by concurrent sequential reads on one file descriptor. Experimental results show that it performs much better than legacy Linux readahead: sequential reading intermixed with random ones are faster by 29%; I/O throughput of interleaved streams could be 4~27 times better, with application visible I/O latencies improved by up to 35 times. This demand prefetching algorithm has been adopted by Linux kernel 2.6.24.
Key words: Linux; operating system; I/O performance; file prefetching; parallel I/O; access pattern

摘 要: 设计并实现了一种按需预取算法,采用更为宽松的顺序性判决条件,并以页面和页面缓存的状态作为可靠的决策依据.它可以发现淹没在随机读中的顺序访问并进行有效的预读,支持对单个文件实例的并发访问而产生的交织访问模式.实验结果表明:相对于原 Linux 预读算法,该算法在随机干扰下的顺序读性能可提高 29%;交织读的性能是传统算法的 4~27 倍;同时,应用程序可见延迟改善可达 35 倍.该算法已被 Linux 2.6.24 内核采用.
关键词: Linux;操作系统;I/O 性能;文件预取;并发 I/O;访问模式
中图法分类号: TP316 **文献标识码**: A

随着内存与磁盘之间的性能差距日渐扩大,磁盘越来越成为数据密集型应用的瓶颈.如何最大限度地挖掘磁盘的性能潜力,长期以来一直都是非常活跃的研究课题.文件预取(prefetching),又称预读(readahead),正是最重要的磁盘 I/O 优化技术之一,并已成为现代操作系统的一项必备功能.工作于系统内核的预取算法实时地监测各应用程序的读请求序列,据此预测即将访问的数据页面,并提前将其批量读入缓存.预取算法与页面替换算法、脏页面写回策略等一起共同构成操作系统内存页面缓存(page cache)的管理框架.

* Supported by the National Natural Science Foundation of China under Grant No.60774038 (国家自然科学基金); the National High-Tech Research and Development Plan of China under Grant No.2008AA01A317 (国家高技术研究发展计划(863)); the Intel Research Council Project under Grant No.4507345522 (英特尔研究委员会项目)

Received 2008-06-07; Accepted 2009-02-24

© 中国科学院软件研究所 http://www.jos.org.cn

预取技术有助于改善 I/O 的两项主要性能指标[1]:首先,异步预取(asynchronous readahead)可将所需的数据提前准备就绪,从而对上层应用程序隐藏磁盘 I/O 延迟;其次,较大的预取粒度有助于提高磁盘的有效利用率和 I/O 吞吐量.一个典型的磁盘 I/O 包含两个基本操作时间 T_a,T_d,即平均访问时间(average access time)和数据传输时间(data transfer time).其中,T_d 才是硬盘的有效利用时间.记 R 为磁盘的持续数据传输率(sustained transfer rate),S 为 I/O 大小,则近似有 $T_d=S/R$,磁盘有效利用率 $U=T_d/(T_a+T_d)=S/(RT_a+S)$,I/O 吞吐量 $B=RU=RS/(RT_a+S)$.显然,I/O 尺寸越大,数据传输时间所占的比重就越长,磁盘的有效利用率也就越高.若取典型值 R=80MB/s,T_a=8ms,则可根据上述公式画出 T_a,T_d,U 三者与 S 的关系,如图 1 所示.

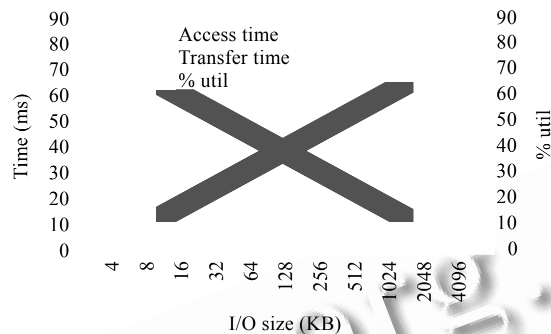


Fig.1 Disks can be better utilized with larger I/O sizes

图 1 大的 I/O 粒度有助于提高磁盘的有效利用率

一般的应用程序都倾向于使用较小的读缓冲区,比如 4KB~32KB.在内存映射文件(memory mapped file)这一便捷、高效的 I/O 方式中,文件访问的粒度仅为一个页面(通常是 4KB).这种小粒度的读 I/O 可引发大量的磁盘寻道操作,造成磁盘有效利用率低下.磁盘的最佳使用方式是进行大的顺序访问.在图 1 中,1MB I/O 的磁盘有效利用率是 4KB 的 100 倍.预读取可以帮助把高层应用的小粒度读请求转化为底层设备的大粒度预读请求,从而提升系统的 I/O 吞吐量.

两个技术趋势给预读算法带来了新的需求和挑战.首先,磁盘寻道的相对代价在持续扩大.在过去的 15 年中,得益于磁记录技术的变革和磁盘转速的提升,磁盘的持续传输率提高了 90 倍;但是访问延迟仅仅降低了 3~4 倍[2],并且因为物理机械的限制,很难再有改进.因而,预读作为一种减少寻道次数的有效技术,其重要性与日俱增.与此同时,这要求进行设计理念的转变,把"避免寻道"上升为预读算法的一个主要设计目标.传统预读算法的核心目标是保证高预读命中率以减少不必要的浪费,因而仅对严格的顺序读进行预读.在内存和带宽资源稀缺的年代,这一目标有其合理性.但随着这两类硬件资源的极大丰富,预读不命中的代价大为降低,因而可以并且应当采用更贪婪的预读策略,对一些非严格顺序 I/O 负载也进行预读.

其次,I/O 的并发度越来越高.随着多处理器、多核、多线程成为提升计算性能的主要手段,计算机系统设计的关注焦点正在从单个线程的执行性能转移到多个甚至大量线程的并行运行上,开发可以在数量持续增长的并行处理器上高效运行的应用程序就变得非常重要.与此同时,软件并行化也意味着 I/O 并行化.预读算法能够在很大程度上增加并行 I/O 的效率.反过来,I/O 的并发形式和并发度也给预读算法带来了挑战:当多个相同或不同类型的并发 I/O 流交织在一起时,如何有效地进行识别和预读?如何在不同的 I/O 并发度的情况下,保证预读算法的时空效率?

针对上述现实需求,我们提出并实现了一种按需预取算法.该算法的主要思路是:适当扩展原有的顺序性条件,并放松对读请求序列严格的模式匹配,以改善预读算法在非经典情况下的鲁棒性和适应性;在传统的预读窗口数据结构基础上引入一个新的预取页面标记,以便稳定地跟踪和高效地识别交织的顺序流;将内核虚拟内存子系统维护的页面缓存视为预读窗口的另一种表现形式,在后者的状态量被其他并发读者覆盖时,仍然可以通过快速扫描页面缓存的相关页面索引重新估计出原窗口的位置信息.

© 中国科学院软件研究所 http://www.jos.org.cn

本文的贡献主要包括如下几个方面:

- 明确了预取算法所面临的新情况和新挑战,分析了传统算法的不足之处,进而给出了解决思路;
- 分析和对比了新老两种算法的设计原理和理念、在若干情境下的行为动态及其对性能的影响;
- 提出了一种基于页面状态的按需预取算法,具有简单性、鲁棒性和更好的性能;
- 提出了根据页面缓存恢复预读窗口的新机制,以简单优美的算法、零空间代价和极小的时间复杂度支持交织读的预取;
- 设计实验验证了新算法对受干扰和交织的顺序访问的有效性.

1 相关工作

预取是一种广泛应用于计算机存储体系各个层面的数据访问优化技术.其中,对磁盘、光盘、固态硬盘(solid state disk,简称 SSD)以及磁盘阵列、存储网络和网络文件系统的预读取是提升 I/O 性能的有效手段.I/O 预取有两个基本类型:对应用程序透明的启发式(heuristic)预读和由应用程序导向的通知式(informed)预取.启发式预读算法观测程序的历史访问记录,分析其 I/O 特征,独立自主地预测并预取即将被访问的数据块.它最成功的一个应用范例是顺序预取(sequential prefetching):顺序读作为最典型和最容易预测的一种访问模式,对它的自动识别和预读已经成为现代操作系统和存储设备的一个标准功能.除此之外,也有研究者尝试挖掘并利用文件之间[3−6]或数据块之间[7]的相关性展开预读.包括 GNU/Linux,Mac OS X 和 Windows 在内的很多操作系统也利用程序及其启动依赖文件之间的强关联性,通过预加载来加快系统和程序的启动速度.

启发式预读主要适用于一些简单可预测的 I/O 行为.更复杂的 I/O 负载类型则需要通知式的预取机制,即要求应用程序通过特定的 API 明确告知预取算法自己的 I/O 特性或即将进行的 I/O 操作.例如,GNU/Linux 操作系统就提供了包括 *madvise*()和 *posix_fadvise*()在内的预取 API.应用层可以通过这类接口主导预读[8−10].由于这通常需要在程序中引入额外的复杂性,其应用仅见于为数不多的程序.为了寻求一种自动为应用程序加入预取指令的普适方法,Chang 等人提出了预执行程序[11,12],Brown 等人提出了编译器分析[13]的思路.

近年来,Papathanasiou 等人建议采用更加贪婪的预取策略[14].我们的研究和实践经验都支持这一论点.虽然本文算法引入了低预取命中率的风险,然而在被 Linux 采纳并广泛发布之后,迄今尚未收到由此导致的关于性能下降的负面反馈.它带来的收益是明显的——增进了算法对随机读的抗干扰能力以及对交织读的支持.

缓存是另一种被广泛应用于金字塔型层级存储体系的 I/O 优化技术.其中,在内核中管理页面缓存的算法主要是经典的 CLOCK/LRU 页面替换算法及其变种.由于包括 Linux 在内的多数内核都把预取页面放在页面缓存中统一管理,因而建立起了预取算法和缓存算法的联系:页面缓存的主体由文件的数据页面构成,并由预读算法加载;当页面被加载到页面缓存之后,转由页面替换算法负责它们的老化(aging)和回收(reclaim).因而,一个的文件数据页面的生命周期通常是(预取算法)加载-(应用程序)访问-(页面替换算法)回收.当系统内存缓存不足时,可能导致预读抖动(readahead thrashing),即被预取的数据页面在被访问前就被回收.按需预取算法中的预读触发条件能够保证预读窗口中的页面被回收后,及时地回退和重新建立新的预读窗口,从而有效地从预读抖动中恢复.进一步的研究包括:综合考虑预取与缓存[8,9,15−18]、动态调整预取缓存[19]以及采用更智能的算法动态地调节预取大小[20−22].

在存储介质方面,有两个正在发生的重要进展:大容量动态内存和闪存的普及.或许有人会认为,大容量内存的出现降低了预读技术的重要性,因为可以缓存的内容多了,自然就减少了 I/O 及其预读的可能性.这种担心只是反映了变化的一个方面.与内存容量增长相呼应的是硬盘以及数据集的同步增长,I/O 密集型的应用无论是广度还是强度都在拓展和加强,数字信息世界的总体磁盘 I/O 数量正在快速地增加.同时,内存的极大丰富和磁盘带宽的快速增长,以及与此构成鲜明对比的磁盘访问延迟的严重滞后,这些因素正在极大地提升预读的效果和贪婪预取策略的相对收益.

闪存作为一种新兴的存储介质,它相对磁盘最突出的优点是没有机械装置,因而访问延迟小、抗震性强,非常适合于以随机读为主的负载.它在移动及嵌入式设备上拥有广阔的发展空间.但由于存储容量的限制,闪存在

© 中国科学院软件研究所 http://www.jos.org.cn

桌面及企业存储领域更多地将作为一种与磁盘互补的介质共同存在.它所擅长的随机和小文件负载类型原本就不是磁盘及其预取算法擅长处理的,而后者所擅长的顺序负载往往具有大数据量的特点,因而也不适合使用闪存.随着闪存容量的提高和造价的降低,固态硬盘将在某些领域成为磁盘的替代者.SSD 一般由一组可以并发传输的闪存芯片构成,因而仍然需要较大的 I/O 尺寸才能充分发挥其性能.这正是预读算法所能提供的.

2 预取算法

2.1 数据结构

预取算法需要维护两种类型的状态量:读历史和预读历史.理论上,完整的读请求历史记录能够最好地反映程序的访问特征和准确地预测未来的数据需求.但是,考虑到算法的复杂性、时效性以及存储空间的开销,历史记录并非多多益善.事实上,当预取命中率不再成为最高目标时,最少的状态量反而有助于提高算法的鲁棒性和适应性.对于顺序预取来说,只需知道上次以及本次的读请求即可进行最简单的顺序模式匹配.我们选取的读历史记录是一个三元组{*prev_offset*,*offset*,*read_size*}.它们分别表示上一次访问的页面在文件中的偏移量、当前访问页面的偏移量以及从当前访问页面算起的请求页面数.

2.1.1 预读窗口和流水线预读

每当预取算法决定要发出一个预取请求时,都会以一个称为预读窗口(readahead window)的数据结构来记录.预取窗口是一个二元组{*start*,*size*},它们分别表示预读请求的起始页面索引和页面数.预读窗口既是本次预取的决策结果,又是下次预取的决策依据.Linux 2.6 的核心预取状态包括两个预读窗口:current 窗口和 ahead 窗口.维护两个窗口是为了实现预读流水线(readahead pipeline):当应用程序在 current 窗口中访问数据并逐步推进的同时,内核在 ahead 窗口中执行异步预取 I/O,以便将数据提前加载就绪.

Linux 2.6.24 及其之后的版本采用了本文提出的按需预读算法.它采用了简化的数据结构,如图 2 所示.其中:*start* 和 *size* 构成一个预读窗口,记录了最近一次预取请求的位置和大小;*async_size* 指示了异步预取的位置提前量,即前方还剩余多少未访问的预取页面时启动下一次预取动作;*prev_pos* 记录了最后的读位置,它可用于简单的顺序性测试.

```
struct file_ra_state {
    pgoff_t    start;        //where readahead started
    int        size;         //number of readahead pages
    int        async_size;   //do async readahead when there are only so many pages ahead
    loff_t     prev_pos;     //cache last read() position
};
```

Fig.2 Data structure for readahead states

图 2 预读状态的数据结构

新算法只维护一个预读窗口.原 ahead 窗口的功能转由独立变量 *async_size* 来实现.后者显式地给出了实现预读流水线所需的关键参数,因而更为简单、明确.这一改变也实现了预读大小和提前量的解耦.一个可自由调节的 *async_size* 将允许预取算法进行自适应的预取时间控制,或者允许用户对一些 I/O 延迟不敏感的应用关闭异步预取,以节省预读页面的缓存占用时间.

2.2 算法框架

传统的预取思路是,在线监控所有的文件访问请求,并以此为基础进行预取决策.Linux 就是一个典型的例子.它包含一个预取例程,在每次程序发出读请求时,系统会首先调用此例程,以检查是否需要预读.这一算法除了进行正常的顺序性匹配及预读之外,还需要判断和处理多种异常情况,包括预取功能是否被禁用,磁盘设备是否太忙而不宜立即进行异步预取,文件是否已经被缓存不需要预取,是否面临过大的内存压力而需减小预取大小,下一个预取的时机是否已经到来,等等.上述逻辑交织在一起,使预取算法变得相当复杂.这对功能扩展和问题解决非常不利.

© 中国科学院软件研究所 http://www.jos.org.cn

在图 3 中,我们引入了一个新的预取框架,把预取算法在逻辑结构上拆分成两大部分:监控/触发部分和匹配/执行部分.监控部分嵌入在程序读请求的响应例程中.它在访问每个文件页面之前,检查此页面是否满足预取的触发条件.匹配部分在逻辑上由一组独立的判决模块组成,每个模块匹配并处理一种访问模式.图 3 中的算法框架支持顺序读和随机读两种访问模式,其中,顺序读又可分初始的(initial)、后继的(subsequent)以及交织的(interleaved)这 3 种情况分别加以处理.如果要支持一种新的访问模式,则只需简单地在匹配部分新增一个相应的模式匹配和处理模块即可.预取逻辑因而变得清晰、简洁和高效.

```
1   read:
2       for each page
3           if page not cached
4               call readahead
5           if test-and-clear PG_readahead
6               call readahead
7       save last read position
8   readahead:
9       if is async readahead and queue congested
10          return
11      if sequential
12          setup initial window
13      else if hit PG_readahead
14          recover window if necessary
15          ramp up and push forward window
16      else
17          read as is; return
18      submit readahead io
19      mark new PG_readahead page
```

Fig.3 Demand readahead algorithm

图 3 按需预读算法

2.3 预取触发条件

传统的预取算法没有触发的概念,或者说,其算法例程无条件地被每一个读请求触发,全程执行严格的模式匹配.在理论上,这样可以使预取算法有机会获得完整的访问信息,从而提高模式识别和预测的可靠性.但实践证明,这反而导致了两个方面的问题:(1) 模式匹配结果的置信度虽然提高了,但也因而变得过于保守和脆弱,无法支持生产环境中多样化的负载类型;(2) 预取例程包含了所有功能,并且被过度频繁地调用,使其变得复杂和低效.

本文算法引入了触发的概念.它是后文中将要叙述的一系列逻辑简化和功能增强的基础.仅当如下两类页面被访问时,才触发预取动作:

(1) 缓存缺失页面(cache miss page).此时需要立即进行磁盘 I/O 加载当前页面,与此同时,应用程序将被临时挂起等待 I/O.此时应当调用预取例程,确定可能被访问的临近页面,进行同步预读.

(2) 预取标记页面(PG_readahead page).此标记是在上一次预取 I/O 中设置的,见到它则意味着进行下一个预取 I/O 的时机已经来到,因而应立即调用预取例程,进行异步预读.相对于前一触发条件,这一触发条件的设置有助于减少乃至消除应用程序的 I/O 阻塞时间.PG_readahead 标记必须在它成功触发预读的同时清除,以避免重复触发的发生.

值得注意的是,我们选择的触发对象是“读页面”.传统算法以“读请求”作为预读决策的基本依据,这带来了不少的问题:首先,读请求的单位是字节,而预取 I/O 的最小单位是 1 个页面,最大单位是 *max_readahead* 个页面.各种负载的读请求大小千差万别,既可以一次读 1 个整数,比如 4Byte,也可以一次性请求发送(sendfile)一整个大文件,比如 4GB.在第 1 种情况下,4KB 的访问量会导致无意义地重复调用 1 024 次预取例程,而在第 2 种情况下,超大的读请求必须被分割成 *max_readahead* 个页面的单元来分次进行预取决策.首先,以页面作为触发对象,可以自然而然地避免这些问题.其次,基于页面的触发也使 *async_size* 规定的预读提前量得以精确地实现.在传统 Linux 预读算法中,异步预读的触发条件是当前“读请求”跨入 ahead 窗口的范围.由于读请求大小的不确定性,

© 中国科学院软件研究所 http://www.jos.org.cn

使得其预读提前量是在一个[$max_readahead$,$2\times max_readahead$]范围内的一个动态值.最后,新的触发条件还能更好地支持重试读(retried reads)和预读抖动(readahead thrashing)等异常情况[23,24].

2.4 顺序性的判定

2.4.1 强时空连续访问

传统算法对顺序文件访问模式的认定包含时间和空间两个方面的要求.它对顺序性的基本定义如下.

定义 1. 假设对一个打开的文件描述符(file descriptor),在它上面进行的一组时间上连续的读请求序列为 $R_{m,n}=\{(pos_i,count_i)|i=m,m+1,\dots,n;n>m;m\geq 0\}$.如果其中任意的两个相邻读请求都满足条件 $pos_{i+1}-pos_i=count_i$, $i=m,m+1,\dots,n-1$,则称 $R_{m,n}$ 为一个时空连续读序列.

定义 2. 如果 $R_{m,n}$ 是一个时空连续读序列,而 $R_{m-1,n}$ 和 $R_{m,n+1}$ 都不是(如果它们存在的话),则称 $R_{m,n}$ 为一个最大时空连续读序列.

在传统算法中,每一个最大时空连续读序列都对应一个最大时空连续预读请求序列.假设 $R_{m,n}$ 的预读序列为 $P_{m,n}=\{(offset_j,size_j)|j=0,1,2,\dots,q\}$,其中,$offset_j$ 和 $size_j$ 的单位不是字节而是页面,内核进行预读和缓存的最小单位.值得注意的是,$P_{m,n}$ 和 $R_{m,n}$ 并不是简单的包含关系,预读的起始页面索引与读序列的位置关系有时是 $offset_0=round_up(pos_{m+1}/page_size)$.这是因为通常只有当看到 R_{m+1},即序列中的第 2 个读请求时,算法才能判断它为一个时空连续读序列,并进行预读.另外,如果 pos_{m+1} 不是指向一个页面的首字节,则表明它所在的页面也包括 R_m 的全部或部分数据,因而此页面刚刚被访问并缓存,所以预读应开始于下一个页面.综上所述,预读区间经常不能包含顺序读序列的第 1 个读请求.为了弥补这一缺憾,预读算法特别对以下两种在实际负载中普遍存在的顺序文件访问情况展开适当超前的预取.

定义 3. 记第 i 个读请求 $R_i=(pos_i,count_i),i\geq 0$,如果 $count_i>max_readahead$,则称 R_i 为一个超大读请求.

超大读一般是与 *sendfile*()系统调用相伴生的,后者被广泛应用于 FTP,HTTP 等网络文件服务器,用于进行高效的零拷贝(zero-copy)文件下载服务,因而具有极强的顺序性特征.在传统预取中,一个超大读请求会立刻触发相应的预读和异步预读动作.

定义 4. 如果第 1 个读请求 $R_0=(pos_0,count_0)$开始于文件的首字节,即 $pos_0=0$,则称其为首次首部读请求.

一个首次首部读请求,通常意味着应用程序将要对文件进行顺序扫描,因而很有必要在所请求的数据之外适当地多预取一些后继页面.

综上所述,传统的顺序性判定条件为表 1 中的任意一种.其中,$prev_offset$ 指向最近一次被访问的页面,$prev_offset==-1$ 表示文件尚未被访问的状态.表中的非对齐读情况出现于前后两个读请求 R_a,R_b 的交界位置不是正好位于文件数据页面的交界位置的时候,此时,前一个读请求 R_a 最后访问的页面索引 $prev_offset$ 与后一个读请求 R_b 访问的起始页面索引 $offset$ 是相等的.

Table 1 Traditional sequentiality criterions

表 1 传统的顺序性判定条件

Criterion	Case
$read_size>max_readahead$	Oversize read, a common case in *sendfile*() calls
$offset==0$ && $prev_offset==-1$	Initial read starts from start of file, sequential reads may follow
$offset==prev_offset$	Consecutive reads that are not aligned to page boundary
$offset==prev_offset+1$	Trivial sequential reads that are consecutive in both time and space

2.4.2 空间连续访问

定义 1 所规定的顺序性条件简单明了并且易于实施,但是过强的约束也限制了它的应用范围.当有多个读者在同一个文件实例上各自进行顺序访问时,它们的读请求将会交织在一起.为此,我们引入流的概念,以处理这种多个空间上连续但在时间上交织的访问形式.

定义 5. 记一个文件的数据页面集合为 $F=\{F_i,i=0,1,2,\dots,n-1\}$,其中,$n$ 是这个文件的数据页面数.对该文件按时间排列的页面访问序列为 $E=\{E_j,j=0,1,2,\dots,m\}$.如果在 E 中从前往后抽取的一个页面访问子序列恰好能顺序

© 中国科学院软件研究所 http://www.jos.org.cn

相连构成一个页面区间 $F_{a,b}=\{F_i,i=a,a+1,\ldots,b,0\leqslant a<b<n\}$,并且前、后页面在页面缓存中的生命周期互相交叠,则称此页面访问子序列为一个空间连续文件访问流,简称流(stream).如果 $F_{a,b}$ 是所有包含对页面 a 的某次访问的流中长度最大的,则称 $F_{a,b}$ 为一个最大流.

在考虑并发流的情况下,表 1 中的传统顺序性规则大体上仍然适用.在其基础上,本文通过引入如下两条算法修正,即可实现对并发交织读的有效识别.

(1) 在传统算法中,顺序性规则被应用于每一个读请求,任何一个随机读都会导致当前顺序流的终止;而本文算法仅在发生页面缺失时执行表 1 中的顺序规则匹配,因而具有优良的抗干扰性.如果一个程序在进行顺序文件访问的过程中夹杂了一些不相关的随机访问,则在传统算法中,时空连续读序列及其预读序列在任意时刻都会被这些随机访问打断.新的算法则仅在决定是否进行初始预读时才依赖于这种时空连续性.

(2) 在被预读标记触发的情况下,无条件进行异步预取.这一规则保证一旦开始了对一个流的预读,这个预读序列就不会被不相关的随机读或交织读者打断.预读还是可能会因为缓存命中而停止,这是预期的正常行为.因此在这一规则下,我们能够预期在交织读的情况下仍能获得稳健的预读行为.但是在另一方面,它也可能导致一些不适宜的预读,对预读命中率造成负面影响.在理论上,可以造出这样的一个稀疏读序列,它的每次访问正好命中预读算法所生成的 PG_readahead 页面标志,此时,这一规则所能造成的最低命中率是 1/*max_readahead*.然而,其发生概率极小:首先,这种情况不会发生在跨步读(stride reads)模式中,因为跨步读是等间距的,而 PG_readahead 页面在预读的起步阶段是非等间距的(见下一节);其次,对一个文件中的均匀随机读模式,连续 i 次随机访问都恰好命中预取标记页面的概率为 $1/P(n,1)^i$,其中,n 为文件的页面数.当 *max_readahead*=32(128KB), n=32768(128MB)时,最低命中率等于 3%,并且它维持连续 i 次的概率为 3e−5i.这种可能性在实际负载中可以忽略不计.

2.5 预取窗口的确定

2.5.1 自适应预取大小和提前量

假设一个最大顺序流的预取序列为

$$\{P_j=(offset_j,size_j)|j=0,1,2,\ldots,q\},$$

称 P_0 为首次预取(initial readahead),$P_1,P_2,\ldots,P_q$ 则称为后继预取(subsequent readahead).

因为内核不知道当前的流会在何处终止,所以它对 P_0 采用一个保守的预读大小.如果应用程序继续访问,从而命中之前的预取页面,则后继预读的大小会被逐次扩大,直至达到 *max_readahead*.具体规则如下:

(1) 首次预取时,预取窗口的初始大小根据读请求的大小来决定:

$$size=read_size\times scale_0$$

其中,$scale_0$ 取 2 或 4.

(2) 在后继预取中,逐次倍增预取窗口:

$$size=prev_size\times scale_1,$$

其中,$scale_1$ 取 2.

(3) 限制最大预取量为 *max_readahead*

$$size=\min(size,max_readahead).$$

异步预取的提前量根据如下规则设定:

(1) 预取提前量通常取最大可能的值:

$$async_size=size.$$

(2) 如果是对非超大读的首次预取,则等到下一个顺序读请求时才启动异步预取:

$$async_size=size-read_size.$$

上述规则实现了动态和自适应的预取大小及提前量.它们的主要设计目标是保证程序在任意时刻终止其顺序访问,预取命中率都在一个可接受的水平.

© 中国科学院软件研究所 http://www.jos.org.cn

2.5.2 并发流的识别和状态恢复

在并发流环境下,预取算法需要快速而可靠地识别一个读请求是否属于某个流,并取得这个流的预取状态.图 2 给出了用于记录预取状态的关键数据结构.理想情况下,应当给系统中的每一个流分配一个预取状态数据结构.事实上,Solaris/ZFS 正是这么做的.然而,这将引入动态分配、查找及替换等一系列复杂和耗时的操作.这些操作的时空复杂度会是一个关于"疑似并发流"数目的函数.一个顺序流通常只有当其第 2 个读请求被观察到之后才能被确认,其第 1 个读请求只能标识为一个"疑似并发流".但是,由于无法作有效地区分,随机读模式下的每一个读请求都会导致一个"疑似并发流"的建立,造成大量的额外开销.Solaris/ZFS 对此采取的对策是为一个文件允许的并发流数目设定一个上限,为此,它必须在支持的并发流数目和随机读的开销两者之间作一个权衡.

注意到一个进程打开一个文件(对应一个文件描述符)后,通常只是简单地进行单一流的访问,因而 Linux 选择简单地为每个文件描述符维护一个流的预取状态.这一设计可以高效地处理实际负载中典型的单一流的情况.然而,其代价是无法支持并发流的情况,因为此时单个文件实例上的多个并发流会互相覆盖状态信息.*prev_pos* 被覆盖后,会导致传统的顺序性判定规则误判交织读为随机读.为此,我们在第 2.4.2 节给出了一套有效识别交织读的顺序性判定算法.剩下的一个问题是,预读窗口也可能被其他并发流所覆盖,从而使预读算法无从推断下一次预读的位置和大小.

我们发现,被覆盖的预读窗口状态量是可以被重新估计出来的,因为窗口的相关信息也同时蕴含在页面缓存中.图 4 显示了一个文件的页面缓存的全景.在图中,应用程序命中了上一次预读所设置的 PG_readahead 页面,因而预读算法认定这是一个交织的顺序读.但是,关于这个流的所有状态量都被覆盖了,因而 *start* 和 *size* 都不可用.但是,根据第 2.5.1 节中的异步预读提前量规则,*size* 与 *async_size* 在所有的后继预读中是相等的.而 *async_size* 可以通过扫描页面缓存来重新获得:从当前 PG_readahead 页面开始,往后扫描直到第一个未被缓存的页面,这中间的页面数就是 *async_size*.对于后继预取,这一扫描所经过的区间就是预读窗口的一个精确估计;对于首次预取,这是一个偏小估计,但其影响有限.

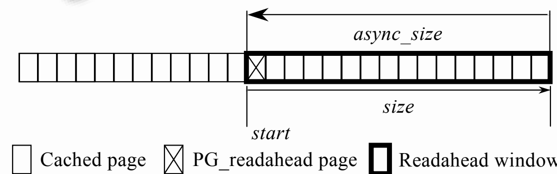


Fig.4 For a typical readahead window, *async_size* equals to *size*, and PG_readahead lies in the first page

图 4 在一个典型的预读窗口中,*async_size* 等于 *size*, PG_readahead 标志位于窗口中的第 1 个页面

3 实例分析

3.1 常规顺序访问

以 UNIX 下最简单的文件操作命令 cp 为例,它从文件的第 1 个字节开始,以 4KB 页面为单位作顺序读.表 2 和图 5 列出了预取算法的前 3 次执行情况.

Table 2 Readahead decisions for command cp

表 2 cp 操作中的文件预读决策

Offset	Trigger condition	Readahead type	Readahead window size (*size*)	Async readahead size (*async_size*)
0	Cache miss page	Initial/Sync	4×*read_size*=4	*size*−*read_size*=3
1	PG_readahead page	Subsequent/Async	2×*prev_size*=8	*size*=8
4	PG_readahead page	Subsequent/Async	2×*prev_size*=16	*size*=16

www.jos.org.cn
© 中国科学院软件研究所 http://www.jos.org.cn

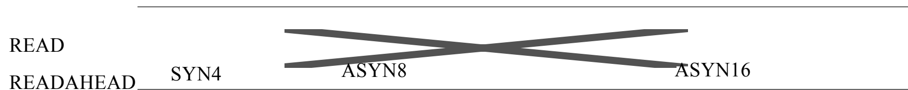
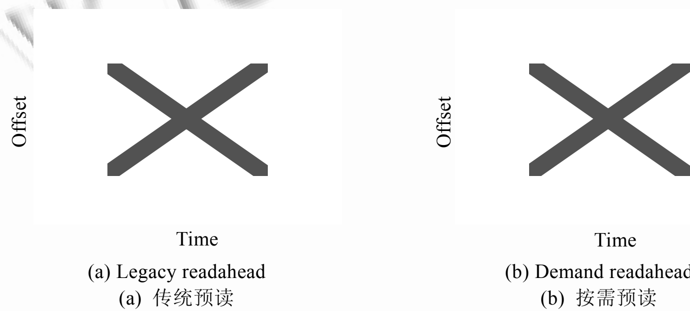


Fig.5 Readahead I/O triggered by the cp command
图 5 cp 命令所触发的预读 I/O

3.2 多个顺序流的并发访问

假设有两个线程 *A*,*B* 共享同一个文件描述符 *F*,并通过 *F* 各自展开独立和不相关的顺序读访问.其中,线程 *A* 访问的页面序列是{1,2,3,…,20},线程 *B* 访问的页面序列是{51,52,…,70}.这两个并发流对 *F* 的读请求将会随机地交织在一起,图 6(a)显示了一种可能的交织方式{1,51,52,53,54,2,3,…}.对于传统的预读算法,对页面 1,51,2,…的读请求不满足顺序性条件,因而会被认为是随机读.其结果是,它感知到的是一些较小的顺序读片段和随机读的混合序列,预读窗口不断地被打开和关闭,但总是难以扩展到理想的大小.

本文算法引入的页面预取标记(PG_readahead 标志位)可以稳定、可靠地标识任意多个并发流.如图 6(b)所示,线程 *A* 的第 1 个读请求(页面 1)仍然会被线程 *B* 对页面 51 的读请求干扰.之后,线程 *B* 对页面 51,52 的访问满足时空连续性条件,因而线程 *B* 的初始预读窗口将会被打开;同时,PG_readahead 标记会被设置.另外,当线程 *A* 连续访问了页面 2,3 之后,它的 PG_readahead 标记也会被设置.线程 *A*,*B* 在接下来的读访问进程中分别命中各自的 PG_readahead 标记,从而触发相应的预读操作.此时,即使没有有效预取状态记录,预读算法也可以借助查询页面缓存来恢复其状态.因而,这两个并发流的预读序列基本上能够与单一流的情况保持一致,从而获得理想的 I/O 性能.



(a) Legacy readahead
(a) 传统预读

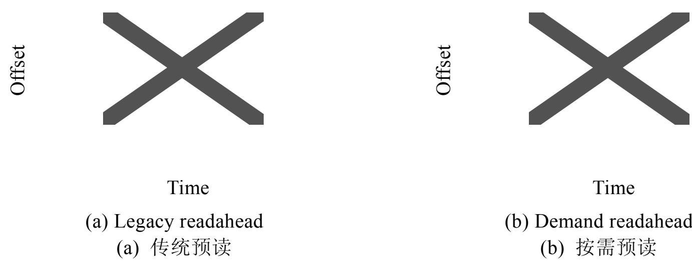
(b) Demand readahead
(b) 按需预读

Fig.6 Interleaved reads by two concurrent streams
图 6 两个并发流产生的交织读

3.3 随机+顺序并发访问

另一种传统算法不善于处理的情况是,在一个顺序流中参杂了一些随机读.如图 7(a)所示,考虑一个访问序列{1,56,72,2,3,4,…},它的主体是一个顺序流{1,2,3,…},但其间也会偶尔访问一些随机的页面{56,72,67, 60,…}.与上一节中并发流的例子类似,在传统算法中,对顺序流的预读会被这些随机读不断地打断,因而效率低下.而在本文算法中,仅对页面 1 访问的顺序性会被打断,而之后的{2,3,4,…}仍会被正确地识别为一个流,对其预读不再会受到随机读的干扰,如图 7(b)所示.如果顺序流开始于第 1 个页面,则预读窗口会被立即打开,因而预读序列将完全不会受到干扰.

© 中国科学院软件研究所 http://www.jos.org.cn



(a) Legacy readahead
(a) 传统预读

(b) Demand readahead
(b) 按需预读

Fig.7 Disturbed sequential accesses
图 7 被干扰的顺序访问

3.4 随机干扰对传统预取算法的影响

传统的模式匹配算法对顺序读的基本匹配条件是 *offset*=*prev_offset*+1.其中,*offset* 是当前访问页面在文件中的偏移量,*prev_offset* 则对应上次访问的页面.如前所述,该条件是保守的.如果应用程序的读序列是{1,2,3,(seek),1 000,(seek back),4,5,6,...},则其中对页面 1 000 的随机读会打断顺序流{1,2,3,4,5,6,...}.预取算法看到的是两个不相关的流:{1,2,3}和{4,5,6,...}.它们将各自触发预取操作,并导致两个不良后果:(1) 预取大小的增长过程被不必要地打断和重新开始,产生较小的 I/O;(2) 对第 2 个流的预取会受到第 1 个流的预取页面的干扰,导致预读缓存命中,产生多余的开销和较小的 I/O.如图 8 所示,流{1,2,3}会触发对页面 2~页面 5 和页面 6~页面 13 的两次预取 I/O.对这个流的预取被随机访问打断后,新的流{4,5,6,...}会重新启动对页面 5~页面 8 和页面 9~页面 16 的两次预取.其中,对页面 5~页面 13 的预取为预读缓存命中,产生不必要的开销;第 2 次对页面 9~页面 16 的预取虽然包括 8 个页面,但是其中的页面 9~页面 13 是预读缓存命中,因而实际产生的 I/O 大小仅为 3 页面.如果预取算法能够不受随机访问的影响,则正常的第 3 次 I/O 的大小应为 16 页面.

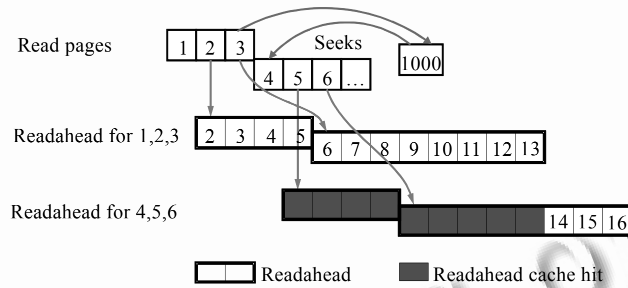


Fig.8 Legacy readahead behavior under a random disturb
图 8 随机访问干扰之下的传统预读行为

4 实验与评价

4.1 实验环境

Linux 内核在其 2.6.23 版本中引入了按需预读的基本框架,包括新的数据结构和预读触发条件,放松了对顺序性的要求.在随后的 2.6.24 版本中,进一步增加了对并发流的支持,从而完全实现了本文描述的按需预读算法.因而,在下文中将直接用标准的 Linux 2.6.22 来作为传统算法的代表实现和测试平台,同时以 Linux 2.6.24 作为按需预读算法的测试平台.两者的最大预读大小均设置为 1MB.实验的硬件配置是:

- Intel(R) Core(TM)2 CPU E6800:内外频率分别为 2.93GHz/266MHz.

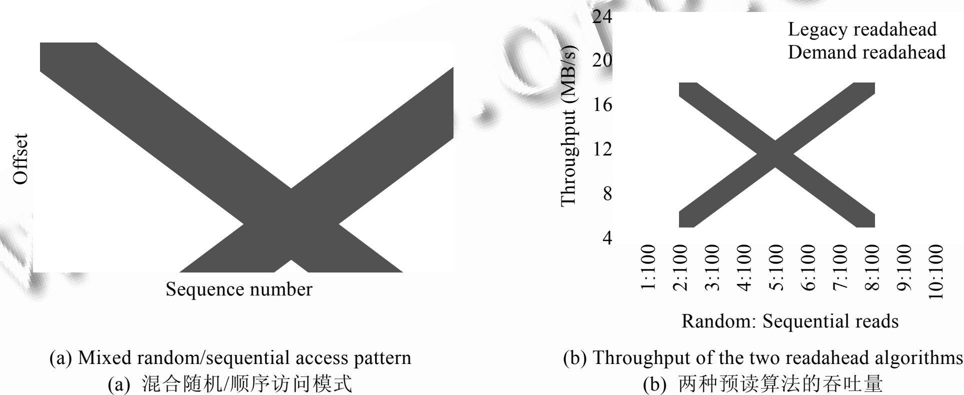
www.jos.org.cn

- 2GB DDR2 内存.
- Hitachi Deskstar T7K250 160GB Hard Drive-SATA II,160GB,7200 RPM,8MB.

在并发 I/O 中,Linux 预读算法能够看到两类访问模式:独立的顺序流和交织的顺序流.例如,Apache 缺省采用多进程模式进行 HTTP 服务,各进程各自独立地打开文件进行顺序访问.这种情况下,文件描述符和 I/O 流之间是一一对应的关系,传统预读算法即可胜任.而 lighttpd 采用单一进程进行 HTTP 服务,文件描述符可被多个客户端共享使用.此时文件描述符和 I/O 流之间是一对多的关系,需要本文所述的算法才能进行有效的预读.为了获得实验的可控性和可重复性,我们首先生成一组独立的 I/O 访问流,然后将它们互相交织在一起,构成一个包含多个并发访问流的读序列,并执行该读序列来检验两种预读算法的性能.

4.2 随机干扰

我们创建了一个 200MB 的大文件,然后对它进行交织的顺序及随机访问.其中的顺序流开始于文件首部,终止于第 100MB;随机读则分布于文件的后半部分.我们生成了 10 个交织的读序列.它们的顺序访问量都是固定的 100MB,而随机访问量从 1MB 到 10MB 逐次增加.每个读请求都访问一个完整的页面,即 4KB.图 9(a)描绘了测试所用的第 1 个读序列,图 9(b)显示了这 10 个读序列执行在两种预读算法下的 I/O 吞吐量.为了避免受到不必要的影响,在实验中关闭了磁盘的内置预读功能.



(a) Mixed random/sequential access pattern
(a) 混合随机/顺序访问模式

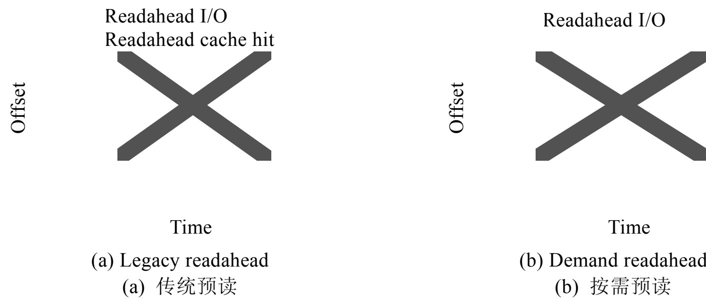
(b) Throughput of the two readahead algorithms
(b) 两种预读算法的吞吐量

Fig.9 Comparison of the readahead algorithms' performance under random disturbs
图 9 预读算法在随机干扰下的性能比较

总体上,两种预读算法的 I/O 吞吐量都随随机访问量的增加而减小.在每一组测试中,本文算法始终对传统算法保持明显的优势.当随机访问量与顺序访问量的比值为 1:100 时,两种算法的吞吐量分别是 17.18MB/s 和 22.15MB/s,新算法的性能高出 28.9%.当这一比例上升到 10:100 时,两者的吞吐量分别降为 5.10MB/s 和 6.39MB/s,新算法仍比传统算法要快 25.4%.

这些性能差异源于两种预读算法在随机干扰下的不同表现.图 10 演示了二者分别为第 1 个访问序列发出的前 1 600 个页面(*start*<1600)的预取请求.在图 10(a)中,传统算法的预读序列会在每次出现随机读时被中断(参见第 3.4 节).原本超前于读位置的异步预读窗口被关闭,然后在重新检测到顺序读时,在当前读位置,从很小的初值 16KB 开始重新增长.由于预读窗口的回退,重新增长初期的很多预读请求直接导致预取缓存命中,而非成为有效预读 I/O.与此形成鲜明对比的是,在图 10(b)中,本文算法的预读窗口稳步增长和推进,没有受到随机读的干扰.

© 中国科学院软件研究所 http://www.jos.org.cn



(a) Legacy readahead
(a) 传统预读

(b) Demand readahead
(b) 按需预读

Fig.10 Comparison of readahead sequences on disturbed sequential reads

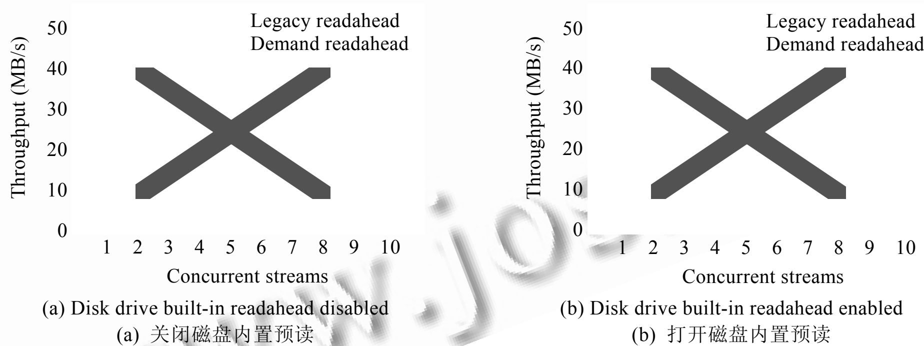
图 10 带随机干扰的顺序读的预读序列对比

4.3 并发流

为了验证并发流的预读性能,我们创建了 10 个顺序流 S_i,i=1,2,…,10,其中,S_i 开始于第(i–1)×100MB 字节,终止于 i×100MB,请求大小为 4KB 的一个连续读请求序列.然后,通过将前 n 个流随机地交织在一起,得到 C_n=*interleave*(S_1,S_2,…,S_n),n=1,2,…,10.其中,C_1 仍然是一个最大时空连续读序列,而 C_n,n=2,3,…,10 则是包含 n 个并发流的一个交织读序列.

访问序列 C_1~C_{10} 在两种预读算法之下所达到的 I/O 吞吐量如图 11 所示.在关闭磁盘预读功能的情况下,传统算法在单一流的情况下吞吐量是 24.95MB/s,稍稍落后于按需预读的 28.32MB/s;当进行 2 个和 3 个流的交织读时,其吞吐量迅速减小为 7.05MB/s 和 3.15MB/s,而本文算法不受影响.当达到 10 个并发流时,传统算法的吞吐量下降到了 0.81MB/s,而本文算法仍维持在较高的 21.78MB/s,是前者的 26.9 倍.

我们同时还测量了磁盘内置预读功能的影响.如图 11(b)所示,磁盘内部预读对性能产生了良好的作用.两个测试平台的单一流吞吐量均达到较高的 51.38MB/s.但是,由于磁盘处理器能力以及缓存大小的限制,它能够有效支持的流数目并不大.随着并发流数目的增长,二者的性能都逐步下降,特别是采用传统预读算法的 2.6.22 内核.当达到 10 个并发流时,传统预读和按需预读算法的吞吐量分别为 12.29MB/s 和 35.30MB/s,后者是前者的 2.87 倍.实际应用中的并发流数往往是数以千计,此时,磁盘的内置预读机制基本不能发挥作用.



(a) Disk drive built-in readahead disabled
(a) 关闭磁盘内置预读

(b) Disk drive built-in readahead enabled
(b) 打开磁盘内置预读

Fig.11 Comparison of readahead performances on interleaved reads

图 11 交织读的预读性能比较

在每组实验的前 100MB 访问中,应用程序所感知的 I/O 延迟的累计值如图 12 所示.该延迟的算式是 $sum(T_t–T_x)$.其中:变量 T_t 是该实验中每个 *read*()系统调用从发出到返回的时间;T_x 是对一个已缓存页面进行 *read*()调用所需的时间,代表 T_t 中所涉及的内存拷贝等非 I/O 等待的部分.随着并发流的增加,传统算法的延迟从单个流的 3.36s 迅速增长到 10 个流的 122.35s,而本文算法是从 2.89s 缓慢增长到 3.44s.在 10 个并发流的情况

© 中国科学院软件研究所 http://www.jos.org.cn

下,传统算法的应用程序可见 I/O 延迟是本文算法的 35.56 倍.这主要归因于传统预读算法的预读窗口很难增长到理想大小(如图 10 所示),并导致两个后果:首先,I/O 偏小,I/O 次数和寻道次数偏多,增加了磁盘 I/O 延迟的总量;其次,预读提前量也偏小,无法有效地向上层应用程序隐藏底层的磁盘 I/O 延迟.

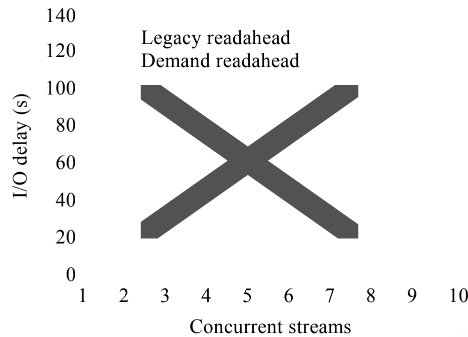


Fig.12 Application visible I/O delays on 100MB interleaved reads

图 12 百兆交织读中的应用程序可见 I/O 延迟

5 结束语

顺序预取是现代操作系统的标准功能.它致力于发现和预测应用程序的 I/O 访问模式,并在此基础上预加载数据页面.预取算法改善 I/O 性能的两个主要途径是,通过提前 I/O 时间来对应用程序隐藏底层的 I/O 延迟,以及通过增加 I/O 大小来获得磁盘的高吞吐量.实现上述目标的前提是预取算法在纷繁的 I/O 访问中准确发现和预测顺序访问模式的能力.

程序行为和系统状态的多样性对算法的适应性提出了很高的要求,并发访问更是对算法能力的一个挑战.这些问题由于磁盘寻道相对代价的增加,多核处理器及并行软件的流行而变得迫切.在借鉴了现有算法大量历史实践经验的基础上,我们采用了以页面及缓存状态为主、模式匹配为辅的创新设计范式,提出并实现了按需预取算法.它在维持并强化原 Linux 预读算法在传统单线程应用中的简单性和高效性的基础上,实现了以零空间开销和不依赖并发度的时间开销有效地支持并发及混合 I/O.本文设计实验证明了该算法的有效性.随着 Linux 2.6.23/2.6.24 的发布,该算法也接受了真实负载环境的广泛考验.

References:

[1] Shriver E, Small C, Smith KA. Why does file system prefetching work? In: Proc. of the Annual Technical Conf. on 1999 USENIX Annual Technical Conf. (ATEC'99). Berkeley: USENIX Association, 1999. 71−84.

[2] Schmid P. 15 years of hard drive history: Capacities outran performance. 2006. http://www.tomshardware.com/reviews/15-years-of-hard-drive-history,1368.html

[3] Kroeger TM, Long DDE. Design and implementation of a predictive file prefetching algorithm. In: Proc. of the General Track: 2002 USENIX Annual Technical Conf. Berkeley: USENIX Association, 2001. 105−118.

[4] Lei H, Duchamp D. An analytical approach to file prefetching. In: Proc. of the 1997 USENIX Annual Technical Conf. Berkeley: USENIX Association, 1997. 275−288.

[5] Pâris JF, Amer A, Long DDE. A stochastic approach to file access prediction. In: Proc. of the Int'l Workshop on Storage Network Architecture and Parallel I/Os. New York: ACM Press, 2003. 36−40.

[6] Whittle GAS, Pâris JF, Amer A, Long DDE, Burns R. Using multiple predictors to improve the accuracy of file access predictions. In: Proc. of the 20th IEEE/11th NASA Goddard Conf. on Mass Storage Systems and Technologies (MSS 2003). Washington: IEEE Computer Society Press, 2003. 230−240.

[7] Li ZM, Chen ZF, Zhou YY. Mining block correlations to improve storage performance. ACM Trans. on Storage, 2005,1(2): 213−245. [doi: 10.1145/1063786.1063790]

www.jos.org.cn
www.jos.org.cn

© 中国科学院软件研究所 http://www.jos.org.cn

[8] Cao P, Felten EW, Karlin AR, Li K. Implementation and performance of integrated application-controlled file caching, prefetching, and disk scheduling. Trans. on Computer Systems, 1996,14(4):311−343. [doi: 10.1145/235543.235544]

[9] Patterson RH, Gibson GA, Ginting E, Stodolsky D, Zelenka J. Informed prefetching and caching. In: Proc. of the 15th ACM Symp. on Operating Systems Principles. New York: ACM Press, 1995. 79−95.

[10] Patterson RH, Gibson GA, Satyanarayanan M. A status report on research in transparent informed prefetching. ACM SIGOPS Operating Systems Review, 1993,27(2):21−34. [doi: 10.1145/155848.155855]

[11] Chang F, Gibson GA. Automatic I/O hint generation through speculative execution. In: Proc. of the 3rd USENIX Symp. on Operating Systems Design and Implementation. Berkeley: USENIX Association, 1999. 1−14.

[12] Fraser K, Chang F. Operating system I/O speculation: How two invocations are faster than one. In: Proc. of the General Track: 2003 USENIX Annual Technical Conf. Berkeley: USENIX Association, 2003. 325−338.

[13] Brown AD, Mowry TC, Krieger O. Compiler-Based I/O prefetching for out-of-core applications. ACM Trans. on Computer Systems, 2001,19(2):111−170. [doi: 10.1145/377769.377774]

[14] Papathanasiou AE, Scott ML. Aggressive prefetching: An idea whose time has come. In: Proc. of the 10th Workshop on Hot Topics in Operating Systems (HotOS). Berkeley: USENIX Association, 2005. 6−10.

[15] Cao P, Felten EW, Karlin AR, Li K. A study of integrated prefetching and caching strategies. In: Proc. of the 1995 ACM SIGMETRICS Joint Int'l Conf. on Measurement and Modeling of Computer Systems. New York: ACM Press, 1995. 188−197.

[16] Dini G, Lettieri G, Lopriore L. Caching and prefetching algorithms for programs with looping reference patterns. The Computer Journal, 2006,49(1):42−61. [doi: 10.1093/comjnl/bxh140]

[17] Gill BS, Modha DS. SARC: Sequential prefetching in adaptive replacement cache. In: Proc. of the USENIX Annual Technical Conf. Berkeley: USENIX Association, 2005. 293−308.

[18] Butt AR, Gniady C, Hu YC. The performance impact of kernel prefetching on buffer cache replacement algorithms. IEEE Trans. on Computers, 2007,56(7):889−908. [doi: 10.1109/TC.2007.1029]

[19] Li CP, Shen K. Managing prefetch memory for data-intensive online servers. In: Proc. of the 4th Conf. on USENIX Conf. on File and Storage Technologies. (FAST 2005). Berkeley: USENIX Association, 2005. 253−266.

[20] Gill BS, Bathen LAD. Optimal multistream sequential prefetching in a shared cache. ACM Trans. on Storage, 2007,3(3): 10:1−10:27. [doi: 10.1145/1288783.1288789]

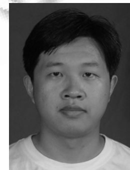
[21] Li CP, Shen K, Papathanasiou AE. Competitive prefetching for concurrent sequential I/O. In: Proc. of the ACM SIGOPS/EuroSys European Conf. on Computer Systems 2007. New York: ACM Press, 2007. 189−202.

[22] Liang S, Jiang S, Zhang XD. STEP: Sequentiality and thrashing detection based prefetching to improve performance of networked storage servers. In: Proc. of the 27th Int'l Conf. on Distributed Computing Systems (ICDCS 2007). Washington: IEEE Computer Society Press, 2007. 64−73.

[23] Wu FG, Xi HS, Xu CF. On the design of a new linux readahead framework. ACM SIGOPS Operating Systems Review, 2008, 42(5):75−84.

[24] Wu FG, Xi HS, Li J, Zou NH. Linux readahead: Less tricks for more. In: Proc. of the Linux Symp., Vol.2. 2007. 273−284.

吴峰光(1977－),男,浙江浦江人,博士,主要研究领域为操作系统,存储系统与 I/O 优化.



徐陈锋(1980－),男,博士,主要研究领域为离散事件动态系统,网络流媒体.

奚宏生(1950－),男,教授,博士生导师,主要研究领域离散事件动态系统,通信网络.

www.jos.org.cn