DiskSeen 预取算法的分析及优化研究

刘 燕 朱春节 王 芳
(华中科技大学武汉光电国家实验室 武汉 430074)

摘 要 计算机存储层次结构是一种典型的金字塔形结构，以平衡计算机对存储系统的两方面需求，即高速处理数据和大的存储容量。然而随着信息技术的飞速发展，计算机处理器和磁盘之间的速度鸿沟持续扩大，因而磁盘访问便成为一个影响计算机系统性能的瓶颈问题。近几十年来，如何减小磁盘访问延迟对整个计算机系统性能的影响，一直是存储领域的热点研究问题。预取技术，通过提前预测I/O请求并将数据读入缓存中，以对上层应用程序隐藏I/O延迟，是缓解这一瓶颈问题的重要技术手段。DiskSeen是一种块级预取算法，通过分析磁盘块的位置和访问时间的联系来提高磁盘访问的顺序性和总体的预取性能。针对DiskSeen算法，文中主要做了以下几方面工作：首先，分析DiskSeen算法的不足之处，据此提出动态控制预取粒度和二次匹配激活历史预取方法，以优化效率；然后，实现了DiskSeen算法及改进后的算法；最后，在模拟仿真实验环境下对算法进行了性能对比测试。实验结果显示，DiskSeen算法能够明显提高缓存命中率并减少平均响应时间，而优化后的DiskSeen算法则可以进一步提升上述两方面的系统性能。

关键词 存储系统，预取，I/O延迟，DiskSeen，优化
中图法分类号 TP302 **文献标识码** A **DOI** 10.11896/j.issn.1002-137X.2017.06.004

Analysis and Optimization of DiskSeen Prefetching Algorithm

LIU Yan ZHU Chun-jie WANG Fang
(Wuhan National Laboratory for Optoetectronics, Huazhong University of Science and Technology, Wuhan 430074, China)

Abstract To balance the demand for high-speed process and large storage capacity, computer storage hierarchy is a typical pyramid-shaped structure. However, as the information technology develops rapidly, the speed gap between computer processor and disk has been widened, which makes the disk access become the bottleneck of the computer system. In recent decades, how to reduce the impact of I/O delay has been a hot issue in storage fields. Prefetching, namely predicting I/O requests in advance and reading the data into cache to hide I/O delay behind application, is a vital technique to alleviate the bottleneck. DiskSeen is a block-level prefetch policy. It can improve the sequential access of disk and prefetching performance by analyzing the relationships between the blocks' locations and access times. Based on DiskSeen, the following work was done in this paper. First, based on the drawbacks of DiskSeen, we proposed dynamic control for prefetch size and active history-aware prefetch in second match to optimize the efficiency. Secondly, we realized DiskSeen algorithm and the optimization of it. Lastly, we conducted comparison test in a simulation environment. The results show that DiskSeen can effectively increase hit ratio and reduce average response time. Furthermore, our optimization can further improve system performance on above two aspects.

Keywords Storage system, Prefetching, I/O delay, DiskSeen, Optimization

1 绪论

1.1 背景

文字是人类社会文明的传承，而计算机的出现使得人们开始以数字化的形式记录文字。计算机由输入单元、输出单元、CPU内的控制单元、算术逻辑单元以及存储器这5部分组成。其中最为重要的就是CPU和内存。CPU是整个计算机的决策机构，其做出判断的数据都源于内存。在整个计算机体系中，基于对数据的容量和高速处理数据的需求，将存储器分为了几个等级。计算机的存储层次结构是一种典型的金字塔形结构，如图1所示。

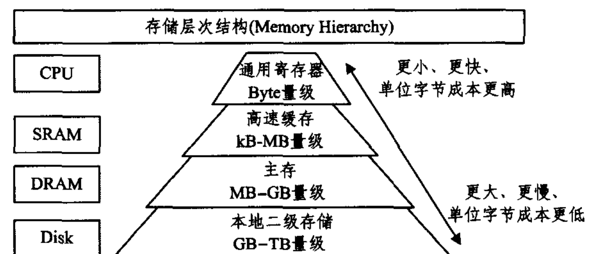


图1 计算机存储层次结构图

到稿日期：2016-11-11 返修日期：2017-01-03

刘 燕 女，硕士生，主要研究方向为并行I/O存储系统和文件系统；**朱春节** 男，博士生，主要研究方向为海量网络存储系统和并行I/O存储系统；**王 芳** 女，博士，教授，主要研究方向为海量信息存储系统、网络存储、并行I/O、容错、并行文件系统、存储能耗等。

由摩尔定律可知,CPU的处理性能在每18个月之后都会翻倍。这个定律经历了时间的验证并获得了社会认可。近几十年来,微处理器的性能正依照着摩尔定律的规律稳步前进。与之形成鲜明对比的是,磁盘容量快速发展,而磁盘的存取速度却增加得十分缓慢,每年仅增长大约8%而已[1]。磁盘的访问时间主要包括3个部分:1)寻道时间,磁头移至相应的磁道上所需时间;2)旋转时间,相应的扇区旋转到磁头下所需的时间;3)传输时间,对数据进行传输所需的时间。由于磁盘内部物理特性的原因,磁盘寻道时间和磁盘旋转时间的减少都会受机械方面的限制,磁盘的访问速度很难再有很大的提高。一般的磁盘访问时间是毫秒级别的,而CPU的处理时间是纳秒级别的。磁盘的访问速度远远慢于CPU的处理速度。而随着处理器速度变得越来越快,两者之间的速度差距也必定会与日俱增,这使得磁盘访问延迟对整个计算机系统性能的影响越来越大。

近几十年来,如何减小磁盘访问延迟对整个计算机系统性能的影响,一直是存储领域着重研究的热点问题。预取技术,通过提前预测上层应用所需数据并将数据预取到缓存,以对上层应用隐藏数据访问以及数据传输的时间,是缓解这一瓶颈问题的重要技术手段。

预取技术主要从以下两个方面来缓解I/O瓶颈问题[2]:1)预取可以在CPU处理当前事务的同时提前对下次将要访问的数据进行预测,然后将数据提前读入缓存,从而在下次读取数据时,如果预取正确,就可以减少上层应用等待数据的时间,使得上层应用感受不到I/O延迟;2)预取可以在一定程度上提高磁盘利用率、整体传输效率和吞吐量。如果频繁地找寻磁道,会浪费很多的时间,而在一次寻道时读取更多将来可能会访问到的数据则有助于提高整体的传输效率和吞吐量。

1.2 研究现状

早期的固定预取算法是指预取当前内容块后的固定m个块。此方法的缺点是预取比较死板,预取效率不高。后来针对磁盘访问的特性出现了顺序预取算法,这类算法可以自适应调整预取粒度,最大化磁盘效率[3]。该算法主要由顺序序列检测、顺序预取和缓存管理3个部分组成。再后来出现了基于应用暗示的预取算法,这类预取算法需要根据用户给出的暗示来进行异步预取。文献[4]提出了一种可以按照用户喜好来选择预取数据的算法。随着人们对大数据的关注度持续增高,后来出现了基于数据挖掘的预取算法。这类算法的关键和难点是在海量的信息中获取用户真正关心的数据,从而针对用户的喜好进行预取。表1列出了几种预取算法的比较。

表1 预取算法的比较

	固定预取	顺序预取	应用暗示的预取	数据挖掘的预取
盲目性	YES	NO	NO	NO
适用范围是否广泛	YES	YES	NO	YES
缓存、预取是否可控	NO	YES	YES	YES
对应用是否透明	YES	YES	NO	YES

大体上可以把预取算法划分为两种:1)启发式预取(heuristic),对上层应用透明,立足于用户的历史访问信息;2)通知式预取(informed),该方法需要由上层应用指导。

启发式预取通常都有3个步骤:1)分析用户的历史访问信息;2)建立数据访问模型;3)执行预取操作。顺序预取就是这种预取方法最为典型且成功的一个范例。启发式预取有以下几种:1)基于时序的分析方法。文献[5]提出了一种使用ARIMA模型的预取方法。该方法采用了时间序列预测和空间马尔科夫链模型预测,可以更加准确地决定预取的时机和预取的块数量。2)基于数据挖掘技术的分析方法,例如文献[6]提出的I/O签名方法,该方法利用预先确定的签名来指导预取操作。启发式预取的对象都相对比较简单,而更为复杂的数据访问模式不易于被启发式预取发现,因而需要使用通知式预取来识别。在进行通知式预取时,需要上层应用提供具有针对性的API,由该API明确指示其即将进行的操作或者I/O特性。

1.3 相关技术

图2示出了常见的4种优化I/O的方法。预取技术虽然各有优劣,但是基本上都包括了使用缓存尽可能地避免I/O,以及异步执行I/O、隐藏I/O延迟的优化手段。

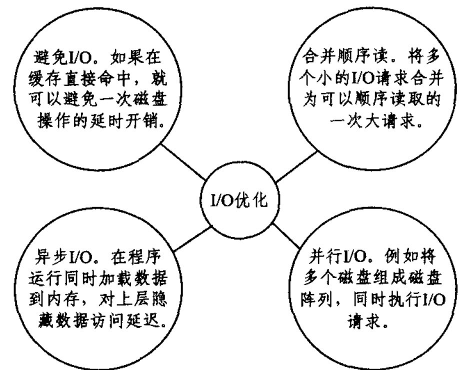


图2 4种I/O优化方法

目前,根据预取算法在存储系统结构中所处的层次,可以将其划分为两类:1)文件级预取算法;2)块级预取算法。

文件级预取算法的代表有NEXUS等。基于加权图的预取算法NEXUS主要是为集群元数据服务器而设计的[7]。

该算法预取元数据而不是文件数据,相对文件数据而言,元数据的优势在于数据量更小,这降低了错误预取代价,同时也可以预取更多的元数据。因此算法可以采用更为激进的预取,但同时也保证了相对较高的预取准确率。文件级预取的最主要的缺陷是预取粒度相对较大,相应地,一旦预取错误,过度预取的代价也就大得多,这会造成缓存区的频繁抖动,与预取的初衷背道而驰。

块级预取算法可以和文件级预取算法相互补充、协同工作,可以更精细地控制预取粒度。按照不同的预取方式,可以将现有的块级预取算法细分为3类:1)基于顺序访问模式的预取算法;2)基于函数拟合的预取算法;3)基于频繁序列挖掘的预取算法。

基于顺序访问模式的预取算法有STEP等。STEP提出了一种新的顺序预取算法,以解决在实际应用中常见的同时运行多个请求而产生多顺序流交叉的问题[8]。当一个新请求到来时,参考最近的历史访问信息来判断是否为一个顺序序列。该算法在预取时维护多个顺序流,在看似杂乱无章的随

机访问中找出隐藏的顺序访问，以此来提升顺序预取的性能。这类算法的主要缺陷是对于已经探测过的顺序流仍会重复之前的探测过程，如果顺序序列重复率很高，该算法的效率反而会更低。

基于函数拟合的预取算法以 UGEP[9] 为代表。数据块之间的相关性是存储缓存、预取、数据布局以及磁盘调度的理论基础，然而在存储层次并不能直接获得数据块间的相关性。UGEP 可以高效地分析用户的访问顺序，并找到块之间的语义相关性。该算法采用统一的设计初始化种群，能够让个体分布均匀。在遗传演进过程中，最重要的就是算子。UGEP 采用了自适应多亲交叉算子。在实验过程中，通过训练一部分负载，获取可以指出块之间量化的相关性的方程式。这类算法的主要问题是函数拟合过程中的开销比较大，预取的准确率相对较低。

基于频繁序列挖掘的预取算法以 C-Miner 为代表。C-Miner 使用了一种数据挖掘技术来发现数据块之间的相关性[10]。该算法使用数据挖掘技术在一系列顺序序列中挖掘频繁序列，进而用这些频繁序列来推断数据块之间的关系。然而，这些频繁子序列只是那些经常一起出现的块集合，不可以直接作为预取的依据。C-Miner 需要将那些频繁出现的子序列转换为关联规则，并在此基础上进行预测。这类算法的主要问题是频繁序列的挖掘开销比较大，对顺序流的预取效果不及顺序预取类算法，同时在进行频繁序列和关联规则转换的过程中也不够灵活。

2　算法的优化

2.1　缓存划分和历史路径

（1）缓存划分

预取通过主动预测上层应用将会需要的那些数据并将其读取到内存中，来减少读盘时间。如图 3 所示，图 3(a) 是原始的系统，而 DiskSeen[11] 用到的是图 3(b) 中的划分方法。从原来的缓冲区中划分出一部分内存作为预取区，用来存储预取的数据。

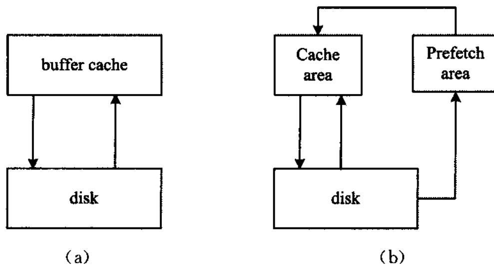


图 3　内存划分

上层应用请求数据时，会先在内存中寻找。如果找到，则直接返回；如果没有找到，则须读盘。在把数据返回给上层应用的同时，也把数据存入缓存区。DiskSeen 若发现有历史路径能够与当前访问重合，就会激活历史感知预取；如果检测到访问的是一个顺序流，就会激活顺序预取。从磁盘中读取预取内容并将其放入预取区。如果下次在预取区命中请求，则不用读盘。在将预取区的数据返回给上层应用的同时，也把数据从预取区转移到缓存区。缓存区的数据采用 LRU 算法来管理，当缓存已满时，将最近最久未被访问的数据淘汰出去。当预取区满时，就把那些从来没有被上层应用访问过的数据淘汰出去。

DiskSeen 包含两种预取算法：顺序预取和历史预取。后者的优先级高于前者。在刚开始访问一个文件时，由于块表中没有历史访问记录，预取的数据大部分都是依靠顺序预取做出预取判断。随着访问次数越来越多，块表中记录的访问历史信息越来越丰富，从而可以依据历史感知预取做出更为准确的预取判断。

（2）块表

磁盘设备接口通常都会提供磁盘的逻辑地址信息，也就是线性化的磁盘逻辑扇区地址，以序列$[0,1,2,3,\cdots,n]$表示的一系列逻辑块编号（LBNs）。磁盘生产商们通常都会尽可能地保证那些逻辑块编号（LBNs）连续的块在磁盘上的物理位置也相邻[12]。

在追踪历史访问路径、寻找匹配的历史路径时会用到“块表（block table）”。块表的结构如图 4 所示。块表可以分为多个层级，最后一级为叶子级（BTE），前面的都是目录级。假如每一个页表项可以存放 100 个条目，那么 LBN 为 $5*100*100+6*100+7=50607$ 的块的访问时间被放入 7 号叶子级条目中，而这个叶子级条目可以从 5 号一级目录和 6 号二级目录中找到。

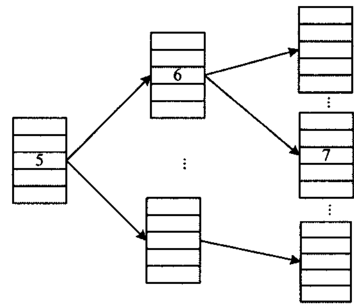


图 4　块表

在 DiskSeen 的设计中，把磁盘块的整个访问序列看成一条块访问流。这个流中的第 n 块的访问时间即为 n，它会被记录到相应的块表的叶子级条目中以表示这个块的访问时间。算法中设置了一个计数器，用来标志块的访问时间。每当接收到一个块的请求时，计数器的值就会增加。当一个块请求的服务完成时，就会把当前计数器的值作为访问时间记录到块表内相应的 BTE 中。每一个 BTE 最多可以记录最近的 4 个访问索引。一个 BTE 的大小是 128 位。每一个访问索引有 31 位，其他 4 位用来表示磁盘块的状态信息。

（3）历史访问路径

在进行历史感知预取时，会涉及“路径（trail）”这一词。在同一条路径中，相邻块的访问时间相差不大，并且这些块都位于磁盘中的某一个相对集中的局部区域。假设块$(A_1,A_2,A_3,\cdots,A_n)$是同一条路径中的块，$0<\text{access stamp}(A_i)-\text{access stamp}(A_i-1)<T$，$|LBN(A_i)-LBN(A_1)|<S$，$i=2,3,\cdots,n$。每一个块对应的块表的叶子级的条目中最多可以存

放最近的4次访问时间,这4个访问时间都可以用来进行上述条件的判断。如果A_1是一条路径的起始块,那么在这条路径后面的所有块的LBN与A_1的LBN的差值的绝对值都必须在S以内,即路径中的块的LBN在A_1的左右两侧且与A_1的距离都不能超过S。这样就保证了路径中的块都在磁盘的某一个集中的局部空间内,预取这样的路径中的块是有效的,而且错误预取的代价会较小。按照那些块的LBN增加的顺序来进行预取,这样也能减少磁盘头移动的时间开销。

图5为访问路径的示意图,访问时间间隔阈值T设置为256。图中有4条路径以A_1为起始块:1条当前访问路径和3条历史访问路径。路径1(A_1,A_3,A_4,A_5)对应于正在进行的连续块访问,这条路径不能激活基于顺序序列的预取,因为没有对块A_2的访问。有2条历史路径与当前路径重复:路径2和路径3,但是路径2只有一部分和路径1重复。一条路径也有可能走相反的方向,比如路径4。

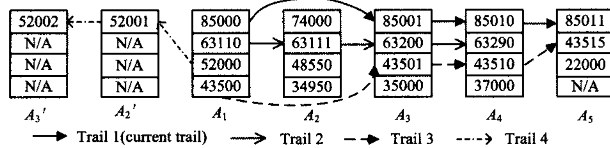


图5 访问路径

由图5可知,若当前路径从块A_1跳转到块A_3,则可以发现2条历史路径:路径2(A_1,A_3)和路径3(A_1,A_3)。当前路径访问到块A_4时,路径2和路径3都包含了块A_4。然而,当前路径访问到块A_5时,只在路径3中有块A_5可以匹配当前路径,而路径2没有包含块A_5。这说明路径2与当前访问路径不匹配,而路径3与当前路径相匹配。

2.2 DiskSeen算法的缺陷

DiskSeen算法利用了具体的磁盘信息(LBNs),从而在很大程度上提高了磁盘访问的性能。然而,DiskSeen算法也存在一些不足,针对这些不足进行优化可以进一步提高DiskSeen算法的性能。

1)没有动态控制预取粒度。在实施预取时,对预取窗口的大小而言,DiskSeen设置了固定的min和max来控制预取粒度,没有考虑不同负载下单次请求的数据块的数目有所不同。若预取数目过小,则不足以提升整体的性能;若预取数目过大,一旦预取失败,则会造成内存数据的浪费和预取数据的浪费,同时增加时间开销,进而不足以体现预取的优势。

2)在进行历史感知预取时,只要找到与当前路径相匹配的4个块就会即刻激活历史感知预取,没有考虑这次预取是否是可信的。如果满足匹配条件就激活历史感知预取,预取无效块的几率就会大大增加。可以把根据匹配的块(A_1,A_2,A_3,A_4)扩展出来的块序列($B_1,B_2,\cdots,B_n$)看成是相互关联的块,即如果出现了匹配序列(A_1,A_2,A_3,A_4),那么接下来就会访问序列($B_1,B_2,\cdots,B_n$)。在DiskSeen的设计中,只要第一次匹配了块序列(A_1,A_2,A_3,A_4),就会预取块序列($B_1,B_2,\cdots,B_n$)。如果匹配序列(A_1,A_2,A_3,A_4)只出现一次,又激活了历史感知预取,一旦预取错误,则会极大地影响算法的整体性能,而且会降低通过历史感知预取进入预取区的数据块的命中率。

2.3 DiskSeen算法的优化

2.3.1 动态控制预取粒度

在DiskSeen算法中,设置了预取的最小值(min)和最大值(max)。用f来监控在当前窗口中命中数据块的个数,如果$2*f<\min$,就取消预取;如果$2*f<\max$,就预取$2*f$个块;如果$2*f\geq\max$,则只预取max个块。这里的min和max的大小是固定的。在基于顺序序列预取的时候,min为8,max为32。在历史感知预取的时候,min为8,max为64。

DiskSeen没有根据请求的大小动态地调整预取粒度,这在一定程度上会对预取的性能产生负面的影响。如果每次请求的块数都很多,假设每次请求48个块,那么最多会预取32块,剩下的16个块没有读入内存,因此还是需要读磁盘,即这次请求没有全部在内存中命中,也就没有带来性能的提高。如果每次请求的块数都很少,假设每次请求4块,而预取粒度最小为8块,那么就不会进行预取,这样也就不能发挥预取的作用。

因而,在DiskSeen的基础上增加一种动态调整预取粒度的方法,即针对不同的请求大小调整预取的粒度。同时,还要设置预取的最小值和最大值,最小值就是预取请求的大小,基于顺序序列的预取的最大值为请求大小的4倍,而历史感知预取的最大值为请求大小的8倍。当预取粒度小于最小值时,取消预取。因为只预取一个请求中的部分块并不能带来整体性能的提高,在预取粒度大于最大值时,只预取max个块。如果预取的块数太多,那么一旦预取错误,所需代价就会很大:1)会造成内存浪费。预取大量的块势必要将一些块淘汰出去,缓存中原本可能有用的数据块淘汰很有可能被出去。2)会造成预取浪费。可能会把刚刚预取而还没有来得及访问的数据块淘汰出去。3)会增加时间开销。预取大量的块需要一定的时间开销,若预取错误还要进行一次读盘操作,使得时间开销增大。因此,设置预取的上限和下限对于提升预取的效率是很有必要的。

2.3.2 二次匹配激活历史感知预取

在DiskSeen算法的原型设计中,只要满足了匹配的条件,就会激活历史感知预取,这在一定程度上降低了预取数据块的可信度。一旦预取错误,很可能会造成缓存数据的浪费和预取数据的浪费,更不能体现预取的优势。

因而,在DiskSeen算法的基础上采取二次匹配激活历史感知预取的策略,使得历史感知预取更为谨慎。假设块序列(A_1,A_2,A_3,A_4)满足匹配条件,块序列($B_1,B_2,\cdots,B_n$)是根据匹配的块序列扩展出来的预取候选块序列。在块序列(A_1,A_2,A_3,A_4)第一次出现时,并不激活历史感知预取,只是将此序列记录下来;只有块序列(A_1,A_2,A_3,A_4)出现第二次、第三次乃至更多次的时候才会认为这个匹配序列是有效且可信的,这时才会激活历史感知预取。这种谨慎的预取方式可以提高使用历史感知预取进入预取区的块的命中率,也使得算法的整体性能有所提高。同时,在匹配序列第一次出现时不做预取操作,如果满足顺序预取的条件就会激活顺序预取。这就相当于提高了顺序预取的优先级,可以认为这种情况下的顺序预取比历史感知预取更加可信。

3 预取流程

3.1 顺序预取

依据磁盘访问的特性，顺序访问可以最大化磁盘效率。因此，在一次磁盘访问操作中顺序读取更多的数据就会获得更大的性能提升，即顺序访问模式可以最大化地开发磁盘访问的性能。基于顺序序列的预取分为两个步骤：顺序序列检测和顺序预取数据。

(1)顺序序列检测

当检测到上层应用请求访问 K 个连续的块时，基于顺序序列的预取就被激活。检测是实时进行的，如果当前访问和上一次的访问构成了一个顺序序列，则判断这个顺序序列的长度是否达到了 K，若达到则预取最后一个请求后面的数据；如果当前访问和上一次的访问不构成一个顺序序列，就从当前访问重新开始记录顺序。

(2)顺序预取

当检测到一个顺序序列之后就开始进行预取，预取时会创建两个窗口，分别是当前窗口和预读窗口，它们的大小都是 min，即可以存放 min 个数据块。然后，在进行后面的磁盘访问的过程中会随时监控在当前窗口的命中率，以 f 来表示在当前窗口命中的块的数目。如果监控到开始请求预读窗口中的数据，就会创建一个新的大小为 $2*f$ 的预读窗口，而现在的预读窗口会成为新的当前窗口。如果 $2*f$ 比最小值还小，就会取消预取，这是因为预取少量的块不能弥补磁盘头移动带来的时间开销，得不偿失；如果 $2*f$ 比最大值还大，那么预取的数目就为最大值，这是因为如果激进地预取大量的块，一旦预取错误，相应的代价也会更大。在预取时，实际读入预取区的块数目可能会比指定的预取数目少，这是因为不会预取那些已经在内存中的块。根据激活预取的请求大小设置 min 和 max。

3.2 历史预取

在基于顺序序列的预取中只用到了当前的访问信息和检测到的顺序访问序列，而在历史感知预取中将会用到块表中更为丰富的历史访问信息。

历史感知预取共有 3 个步骤：寻找历史匹配路径，对历史路径进行扩展和历史预取。

(1)寻找历史匹配路径

在寻找历史匹配路径的过程中，重点是判断当前路径是否和之前的某条路径重合。而是否重合的条件就是两个连续访问的块的访问时间和 LBN 号是否在指定的阈值范围内。假设当前访问从块 A 跳到块 B(A 和 B 的 LBN 差的绝对值须在 S 以内)，如果在块表中块 A 和块 B 的历史访问时间间隔在 T 以内，那么块 A 和块 B 就满足匹配条件。图 6 为寻找历史匹配路径的流程图。

判断是否存在匹配路径的步骤如下：

1)如果当前访问在之前记录的路径范围中，就把当前访问加入到历史访问路径中，并将记录的匹配块数加 1，转 2)；如果当前访问不在之前记录的路径范围中，就从当前访问的这个块开始寻找匹配的历史路径。

2)如果匹配的块数达到了 4，则说明存在与当前访问路径相匹配的历史路径，可以激活历史感知预取。

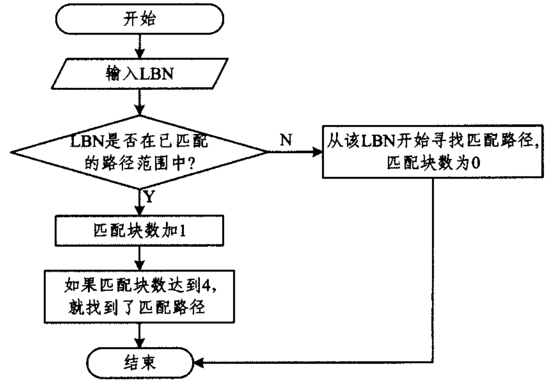


图 6　寻找历史匹配路径

(2)历史路径扩展

在找到了与当前访问路径相匹配的历史路径后，就可以利用已匹配的历史路径来寻找接下来很可能会访问的那些块。假设 A_1 是已匹配的历史路径中的最后一个块，stamp 是块 A_1 用来进行匹配验证的访问时间，访问时间间隔阈值为 T。图 7 为历史路径扩展的流程图。

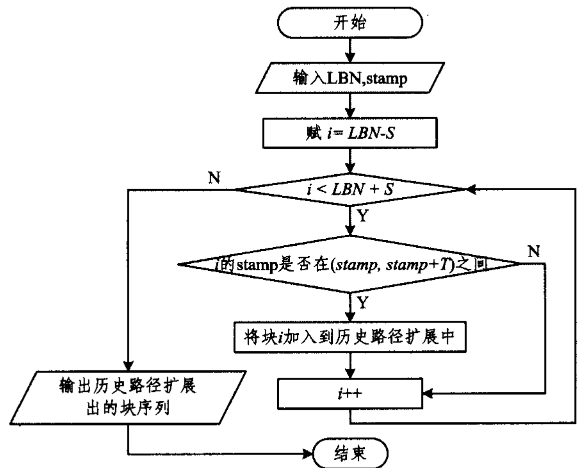


图 7　历史路径扩展

判断一个块是否在当前历史路径扩展中的步骤如下：

1)判断 LBN 是否在 $LBN(A_1)-S$ 到 $LBN(A_1)+S$ 之间，若在，转步骤 2)；若不在，不做处理。

2)判断该块的 4 个访问时间中有没有在($stamp$，$stamp+T$)之间的，若有，这个块就在历史路径的范围中，将其加入到历史路径扩展中；若没有，这个块就不在历史路径的范围中，不做处理。

(3)历史预取

历史路径扩展中存储的是上层应用很可能会访问的那些数据块的 LBN，这些 LBN 就是历史预取的依据。在历史感知预取中同样也维护了两个窗口，分别是当前窗口和预取窗口，它们与基于顺序序列预取中的两个窗口类似。开始进行预取时，先按照顺序读入历史路径扩展中的前 min 块到当前窗口，接下来读入后面的 min 块到预读窗口。这两个窗口的滑动机制也与顺序预取中的窗口滑动机制一样。

如果存在多条与当前路径相匹配的历史路径，就预取在这些历史路径扩展中都有的那些块。在进行历史感知预取

时，至少要有一条匹配的历史路径存在。如果不存在匹配的历史路径，就激活基于顺序序列的预取。

4 仿真实验分析

4.1 实验环境

为了测试 DiskSeen 算法及改进以后的 DiskSeen 算法的性能，在由 C 语言编写的 Disksim 中对无预取、DiskSeen 以及优化后的 DiskSeen 进行仿真测试。Disksim 是一个磁盘系统模拟器，它因为具有效率高、模块多、实验结果准确以及配置多样化的特点而受到广泛存储领域研究人员的青睐。经众多研究人员证实，Disksim 可以真实地再现访问磁盘的过程。

图 8 所示为模拟的实验环境构成图。为了能够真实地模拟 I/O 层的功能，模拟的实验环境由以下几个部分组成：1)用来保存预取数据和原始缓存数据的数据结构；2) LRU (最近最久未被访问)缓存替换策略；3)预取算法，包括 DiskSeen 算法以及优化后的 DiskSeen 算法；4) Disksim(磁盘模拟器)，可以真实地再现磁盘访问。

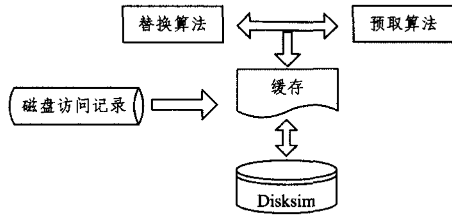


图 8 模拟实验环境

4.2 实验数据

本文共使用了 5 个 IO 负载作为测试数据。这些负载都收集于真实的存储系统，它们是不同场景下真实的应用负载，具有普遍性。同时，很多相关的顶级学术论文中也普遍使用这些负载来验证自己的想法，这也从侧面验证了这些负载的普遍性和有效性。

1) Cello-92 是 1992 年在 Hewlett-Packard 实验室收集的。它捕捉了系统发送的所有低级 I/O 请求。Cello 是一个分时共享系统，HP 实验室的研究者们用它来进行各种实验。

2) Cello-99 和 Cello-92 很相似。唯一的区别在于 Cello-99 是 1999 年收集的，因而其包含的工作负载也更新。

3) financial1 和 financial2 收集的访问记录是在金融机构运行 OLTP 应用的请求数据时，由存储性能委员会提供。

4) websearch2 的数据来自于一个流行的搜索引擎，是真实工作环境下的负载。

以下均以简称代表以上 trace：cello92，cello99，f1，f2，web2。这些 trace 的格式都遵循 SPC trace 文本规范。

4.3 实验性能评价指标

缓存命中是指在缓存中找到了用户所需要的数据，而不用再从磁盘中获取；如果在缓存中没有找到，则称为缓存不命中。假设一次访问中共发出了 N 个请求，req_i 为第 i 次请求。如果缓存命中，$req_i=1$；如果缓存不命中，$req_i=0$。评价 DiskSeen 算法的性能指标主要有以下两个。

(1)缓存命中率(Hit Ratio，HR)，其计算表达式如式(1)所示：

$$HR=\frac{\sum_{i=1}^{N} req_i}{N} \tag{1}$$

缓存命中率是指命中的请求数目在总的 N 个请求中所占的百分比，它是评价预取算法性能的重要指标。缓存命中率越高，说明这次访问在缓存中获得的数据越多，相应地，读盘次数越少。读盘次数越少，在一定程度上就可以减少等待时间。

(2)平均响应时间，即从发出请求开始到接收数据完成所经历的时间(Average Response Time，ART)。缓存命中时可以直接从缓存返回所需数据；缓存不命中时，还要额外增加读盘的时间开销。

4.4 实验结果及分析

4.4.1 命中率的比较

图 9 所示为在 3 种情况即无预取、DiskSeen 和改进的 DiskSeen(动态控制预取粒度、二次匹配激活历史感知预取)下，运行 5 个 trace 后的命中率的比较。

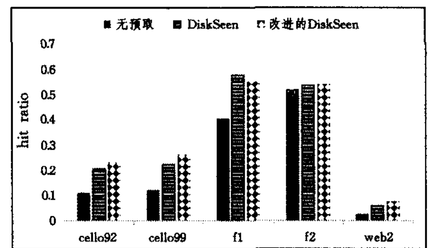


图 9 命中率的比较

图 9 的结果分为 3 组，每组都表示运行相应的 trace 得到的实验结果，即缓存命中率。从实验结果来看，采用 DiskSeen 算法进行预取可以明显地提高缓存命中率，这说明了 DiskSeen 算法的有效性。在进行优化操作后多数 trace 的命中率都有所提高，cello-92 提高了 12%，cello-99 提高了 17%，f2 提高了0.6%，web2 提高了 25%；但是 f1 的命中率下降了 5%。这说明优化操作并不能使得所有的 trace 都提高缓存命中率，针对不同的负载会有不同的效果。但是优化后的 DiskSeen 算法仍然比无预取环境的缓存命中率更高，这说明此优化操作并不会带来严重的负效应。

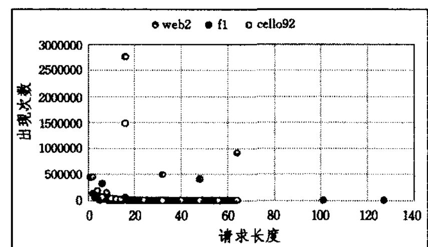


图 10 请求长度分布图 1

从图 10 中可以看出，trace web2 的请求长度分布最规律，集中在(16，32，48，64)这 4 个长度中，因而采用动态控制预取粒度后，其命中率的增幅最大，提高了 25%。而 trace f1 在改进后其命中率反而有所下降，分析其请求长度分布图可以找到答案。从图 10 中可以看出其请求长度集中分布在 20 以内，在 20 到 60 之间出现的次数很少且很分散，而大于 100

的请求出现了两次。可见这些长度很大的请求影响了改进后算法的性能,在将预取粒度调得很大后导致预取错误。这会造成内存浪费和预取浪费,淘汰了原本可能有用的数据,而预取的数据又不是需要的,因而降低了缓存命中率。trace cello92 的请求长度集中分布在 20 以内,在 20 到 60 之间出现的次数很少且很均匀,因而动态控制预取粒度并不会降低命中率。如图 11 所示,trace cello99 和 f2 的请求长度都集中分布在 20 以内。前者的请求长度在 20 到 120 之间有零散的分布,后者则比较连续。

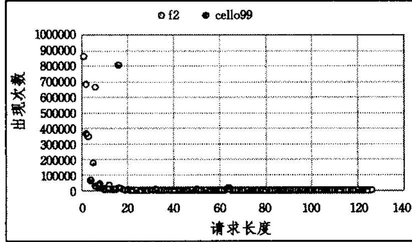


图 11　请求长度分布图 2

综合分析以上几种 trace 的请求长度可以得出以下结论:1)采用动态控制预取粒度的方法可以提高大多数应用负载的命中率;2)请求长度的分布比较集中且有规律的负载的性能提高较明显;3)请求长度范围比较广时,如果出现的相邻请求的长度差值不是非常大,也能一定程度地提高性能,反之可能会使性能有所下降。

4.4.2　平均响应时间的比较

图 12 所示为 3 种情况即无预取、DiskSeen 和改进的 DiskSeen 下,运行 5 个 trace 后的平均响应时间的比较。

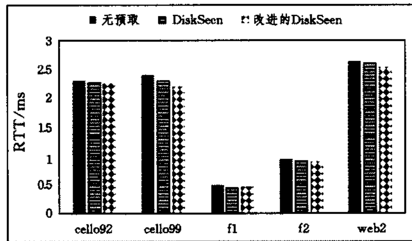


图 12　平均响应时间的比较

图 12 的结果分为 3 组,每组都表示运行相应的 trace 得到的实验结果,即平均响应时间。从图 12 中可以看出,DiskSeen 算法可以明显缩短平均请求响应时间,而改进后的 DiskSeen 算法则进一步缩短了平均响应时间。对比图 12 和图 9 可以发现,命中率越高,平均请求响应时间越短;命中率越低,平均请求响应时间则越长。

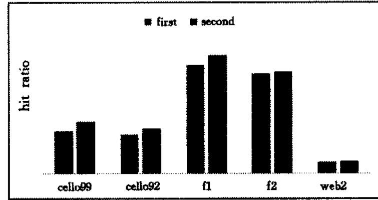
表 2 所列为在 DiskSeen 和优化后的 DiskSeen 算法中,通过历史感知预取进入到缓存中的数据块的命中率的对比。在原始的 DiskSeen 算法中,只要找到了匹配序列,就会立即激活历史感知预取。在优化操作中,采用更为谨慎的预取方式,即在匹配序列第一次出现的时候,不会立即激活历史感知预取,而是将此序列记录下来,如果之后又出现了该匹配序列,就认为这个序列是可信的并激活历史感知预取。在这种情况下,如果检测到顺序序列,就会激活顺序预取。从表 2 中看到,优化后通过历史感知预取的数据块的命中率都有所提升。

表 2　通过历史感知预取的块的命中率对比

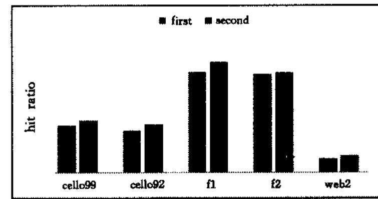
trace	历史感知预取块的命中率	
	改进前	改进后
cello92	0.54	0.58
cello99	0.39	0.47
f1	0.66	0.71
f2	0.70	0.77
web2	0.74	0.78

4.4.3　连续运行两次的对比

图 13 所示为用优化前和优化后的 DiskSeen 算法连续运行两次的缓存命中率的对比。



(a)优化前的 DiskSeen 算法



(b)优化后的 DiskSeen 算法

图 13　连续运行两次的命中率对比

在第一次运行完成后会清空缓存,以保证第二次的运行不会受益于缓存中的数据。但是,在连续运行的过程中,会保存块表中的信息。可以看到,无论是优化前还是优化后,第二次运行的命中率都有明显的提高,这是因为上一次的运行在块表中留下了丰富的历史访问信息。第二次运行时,可以根据块表中的历史信息进行历史感知预取,预取的大多数块都是有用的块。优化后的算法第二次运行时命中率提升的幅度普遍比优化前的大。对于 web2 而言,优化前第二次运行的命中率与第一次运行相比提升了 9.9%,而优化后则提升了 22.4%。连续运行时性能的提高在现实中很有意义,因为用户通常都只会改变一些输入的数据,而不会改变数据集和访问模式。

4.4.4　历史信息的干扰

与当前运行的 trace 相吻合的历史信息无疑会提升历史预取的效率,但如果块表内容是其他 trace 遗留下来的,则很可能会破坏现有的历史感知预取。本文选取 trace f1 和 cello-99 作为对比,观察遗留的历史访问信息给 DiskSeen 算法带来的损失大小。表 3 列出了优化前交替运行两个不同的 trace 的命中率对比。表 4 列出了优化后的对比结果。

表 3　优化之前交替运行两种 trace 的对比

实验	命中率(hit ratio)			
无预取	f1		cello-99	
	0.404		0.118	
1	f1	cello-99	f1	cello-99
	0.579	0.231	0.614	0.272
2	cello-99	f1	cello-99	f1
	0.227	0.603	0.272	0.614

表4 优化之后交替运行两种trace的对比

实验	命中率(hit ratio)			
无预取	f1		cello-99	
	0.404		0.118	
1	f1	cello-99	f1	cello-99
	0.548	0.253	0.581	0.278
2	cello-99	f1	cello-99	f1
	0.256	0.571	0.278	0.581

在这个实验中，交替运行了两个trace。实验1的顺序是f1，cello-99，f1，cello-99；实验2的顺序是cello-99，f1，cello-99，f1。在上一次运行结束后会清空缓存，以保证下一次运行不会受益于缓存中的数据。但是，在连续运行的过程中保留了块表中的信息。对优化后的实验结果进行分析：如果把没有任何历史信息的预取作为参考，历史信息的干扰使得实验1中cello-99的性能下降了1.1%(0.25627 vs 0.25344)，但是在实验2中却意外地将f1的性能提高了4.1%(0.54864 vs 0.57100)。实验1中命中率下降是因为f1遗留在块表中的信息误导了DiskSeen，在cello-99运行的时候判断找到了历史匹配路径，但是却预取错误。在历史预取错误的情况下，需要时间来激活基于顺序序列的预取，这又导致不能充分发挥顺序预取的效果。然而在实验2中，cello-99留下的历史访问信息却使得f1的性能得到提升。这是因为cello-99在磁盘上留下的访问模式仍是顺序的，只会创造更为激进的顺序预取，不会改变顺序预取的本质。在任一实验中，f1和cello-99的第二次运行的结果都和图13中第二次运行的结果很接近，这表明只要块表中还存在一条匹配的历史路径，噪声对历史感知预取的干扰就是微乎其微的。

结束语 随着科学技术的发展步伐越来越快，CPU的处理速度也呈现稳步上升的趋势。然而由于磁盘在物理机械方面的限制，其访问速度很难有大的提高。因此，I/O延迟成为计算机存储系统亟待解决的重点问题。预取，即提前预测应用的请求并将数据读入缓存，以备应用需要的时候快速获取，是解决这一问题的重要技术手段。本文在预取技术相关理论的基础上，重点研究了DiskSeen算法，主要做了以下工作。

(1)提出对DiskSeen算法进行动态控制预取粒度的改进，根据每次的请求大小动态地设置预取的上限和下限。提出二次匹配激活历史感知预取的策略，以提升通过历史感知预取进入预取区的块的命中率。

(2)实现了DiskSeen算法和改进后的DiskSeen算法，并提供了良好的接口。

(3)在模拟实验环境中对算法的性能进行了测试并详细分析了实验结果。

本文在实现DiskSeen算法的基础上对其进行了优化，并在模拟实验环境中进行了性能对比测试。实验结果显示，DiskSeen算法可以明显提高缓存命中率并减少平均响应时间，而优化后的DiskSeen算法则可以进一步提升上述两方面的性能。但在测试以及优化的过程中仍有一些不足，未来将从以下方面继续进行研究。

(1)本文算法的所有测试都只在一块硬盘中进行，未来还需针对多盘交叉访问的情况进行测试。在此之前，要对DiskSeen算法进行一些改变，使其适用于多块盘的环境。

(2)本文对DiskSeen算法及优化后的算法的测试都只是在模拟实验环境中进行的。未来还需要在真实的实验环境中对其进行更完整的测试，以得到更为深入、更有价值的实验结果。

(3)对DiskSeen算法的优化还不够全面。块表在DiskSeen中寻找与当前路径相匹配的历史路径时起了关键的作用。但是块表所占内存很大，磁盘访问的空间局部性又很强，可能只会集中访问某些空间中的块，造成块表空间的浪费。因此，可以采用去除块表，只记录历史访问路径，或者压缩表的方法来进行进一步的优化。

参考文献

[1] GILL B S, MODHA D S. SARC: Sequential Prefetching in Adaptive Replacement Cache[C]// Proc. of USENIX 2005 Annual Technical Conference. Boston, MA, USA: 2005: 293-308.

[2] WU F G, XI H S, XU C F. File prefetching algorithm for concurrent streams[J]. Journal of Software, 2010, 21(8): 1820-1833. (in Chinese)
吴峰光，奚宏生，徐陈锋．一种支持并发访问流的文件预取算法[J]．软件学报，20101，21(8)：1820-1833.

[3] GILL B S, BATHEN L A D. AMP: Adaptive Multi-stream Prefetching in a Shared Cache[C]// Proceedings of the Fifth USENIX Symposium on File and Storage Technologies (FAST'07). San Jose, CA, 2007: 185-198.

[4] PATTERSON R H, GIBSON G A, SATYANARAYANAN M. A Status Report on Research in Transparent Informed Prefetching[J]. ACM Operating Systems Review, 1993, 27(2): 21-34.

[5] TRAN, NANCY, REED D A. Automatic ARIMA time series modeling for adaptive I/O prefetching[J]. IEEE Transactions on Parallel and Distributed Systems, 2004, 15(4): 362-377.

[6] BYAN, SURENDRA, et al. Parallel I/O prefetching using MPI file caching and I/O signatures[C]// Proceedings of the 2008 ACM/IEEE Conference on Supercomputing. Austin, TX: IEEE Press, 2008.

[7] GU P, ZHU Y, JIANG H, et al. Nexus a novel weighted-graph-based prefetching algorithm for metadata servers in petabyte-scale storage systems[C]// Proc. Sixth IEEE Int'l Symp (CCGRID'06). Singapore. 2006: 409-416.

[8] JIANG S, ZHANG X. STEP: Sequentiality and Thrashing Detection Based Prefetching to Improve Performance of Networked Storage Servers[C]// Proc. of ICDCS'07. Toronto, 2007.

[9] CHEN Y, LI F, DU B, et al. A Quantitative Analysis on Semantic Relations of Data Blocks in Storage Systems[J]. Journal of Circuits, Systems and Computers, 2015, 24(8): 1550118.

[10] LI Z, CHEN Z, SRINIVASAN S M, et al. C-Miner: Mining block correlations in storage systems[C]// Proc. of FAST'04. Berkeley, CA, USA, 2004.

[11] DING X, JIANG S, CHEN F, et al. DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch[C]// USENIX Annual Technical Conference. 2007: 261-274.

[12] SCHLOSSER S W, SCHINDLER J, PAPADOMANOLAKIS S, et al. On Multidimensional Data and Modern Disks[C]// Proceedings of the 4th USENIX Conference on File and Storage Technology (FAST'05). Berkeley, CA, USA, 2005.