Chapter 11
Sequential File Prefetching in Linux

Fengguang Wu
Intel Corporation, China

ABSTRACT
Sequential prefetching is a well established technique for improving I/O performance. As Linux runs an increasing variety of workloads, its in-kernel prefetching algorithm has been challenged by many unexpected and subtle problems; As computer hardware evolves, the design goals should also be adapted. To meet the new challenges and demands, a prefetching algorithm that is aggressive yet safe, flexible yet simple, scalable yet efficient is desired. In this chapter, the author explores the principles of I/O prefetching and present a demand readahead algorithm for Linux. He demonstrates how it handles common readahead issues by a host of case studies. Both static, logic and dynamic behaviors of the readahead algorithm are covered, so as to help readers building both theoretical and practical views of sequential prefetching.

INTRODUCTION
Sequential prefetching, also known as readahead in Linux, is a widely deployed technique to bridge the huge gap between the characteristics of storage devices and their inefficient ways of usage by applications. At one end, disk drives and disk arrays are better utilized by large accesses. At the other, applications tend to do lots of tiny sequential reads. To make the two ends meet, operating systems do prefetching: to bring in data blocks for the upcoming requests, and do so in large chunks.

The Linux kernel does sequential file prefetching in a generic readahead framework that dates back to 2002. It actively intercepts file read requests in the VFS layer and transforms the sequential ones into large and asynchronous readahead requests. This seemingly simple task turns out to be rather tricky in practice (Pai, Pulavarty, and Cao, 2004; Wu, Xi, Li, and Zou, 2007). The wide deployment of Linux --- from embedded devices to supercom-
puters --- confronts its readahead algorithm with an incredible variety of workloads.

In the mean while, there are two trends that bring new demands and challenges to prefetching. Firstly, the relative cost of disk seeks has been growing steadily. In the past 15 years, the disk bandwidth improved by 90 times, while the disk access latency only saw 3-4 times speedup (Schmid, 2006). As an effective technique for reducing seeks, prefetching is thus growing more and more important. However the traditional readahead algorithm was designed in a day when memory and bandwidth resources are scare and precious. It was optimized for high readahead hit ratio and may be too conservative for today’s hardware configuration. We need a readahead algorithm that put more emphasis on “avoiding seeks”.

Secondly, I/O parallelism keeps growing. As it becomes more and more hard to increase single-thread performance, the parallel execution of a pool of threads raises to be a new focus. The deserialize of hardware and software means the parallelization of I/O. Prefetching helps parallel I/O performance. In turn it is also challenged by the forms and degree of I/O concurrency: How to detect and keep track of states for multiple I/O streams that are interleaved together? How to maintain good readahead behavior for a sequential stream that is concurrently served by a pool of cooperative threads? How to achieve low time and space overheads in the cases of single thread I/O as well as highly concurrent I/O?

The questions will be answered in the following sections. We will start by the introduction to prefetching with its values for storage devices and roles in I/O optimization. We will give a reference to basic forms of prefetching, discuss the design tradeoffs and argue for more aggressive prefetching policies. We then proceed to introduce the demand readahead algorithm in Linux and explain the differences to the legacy one. At last we demonstrate its dynamic behaviors and advantages by a host of case studies and benchmarks.

**PRINCIPLES OF I/O PREFETCHING**

Bandwidth and latency are the two major aspects of I/O performance. For both metrics, there have been huge and growing performance gaps between disk, memory and processor. For example, the Intel(R) QuickPath(TM) Interconnect (QPI) can provide 12.8GB/s bandwidth per direction for processor-to-processor and processor-to-io data transfers. Today’s DDR3-1333 memory has a theoretical bandwidth of 10666MB/s and response time of 12 nanoseconds, while a Seagate(R) 7200.11 SATA disk has a maximum sustained transfer rate of 105MB/s and average seek time of 8.5ms. Hence the performance gap as of 2009 is about 10 times for bandwidth and 7e5 times for latency. In this section, we demonstrate how I/O prefetching can help fill the performance gaps.

**Storage Devices**

The I/O latency is such a dominant factor in disk I/O operations that it can be approximated by a simple I/O model. A typical disk I/O takes two steps: Firstly, the disk head moves to the data track and waits for the data sector to rotate under it; Secondly, data read and transfer start. Correspondingly there are two operational times: the average access time, whose typical value is 8ms; the data transfer time, which roughly equals to the multiplication of I/O size and the disk’s sustained transfer rate, which averages to 80MB/s for commodity disks today.

In a full I/O period, only the data transfer time makes real utilization of the disk data channel. The larger I/O size, the more time will be spent in data transfer and less time wasted in seeking, hence the more we can harvest disk utilization ratio and I/O bandwidth. Figure 1 reflects this correlation given the above disk I/O model and representative parameter values. One major function of I/O prefetching is to shift a disk’s working point from left to right in the graph, so as to achieve better disk utilization and I/O bandwidth.