Linux PageCache Optimization for Hadoop

Naredula Janardhana Reddy (njana@yahoo-inc.com)
Performance Engineering, Yahoo, Banglore

Abstract

This paper describes a model to manage File Cache (a.k.a Page Cache) smartly for Hadoop nodes to significantly improve hadoop I/O performance. In Linux ,the LRU (Least Recently Used) algorithm is used for page replacement; however it is observed that LRU alone is not suited for typical Hadoop workloads, where most of the files are read once and written once. A selective approach that uses both Most Recently Used (MRU) and LRU based on the file type (HDFS vs. temp) and access mode (read vs. write), optimizes file cache usage. Along with MRU, we have defined and incorporated two related techniques called Prioritized Early Write and Read Advance that improve write and read latencies respectively. These techniques effectively minimize on-disk operations for the short-lived Hadoop temporary files. Additional space for these short-lived files is made available by promptly evicting long-lived HDFS files as soon as they are written and all Hadoop files (temporary and HDFS files) as soon as they are read. We implemented the above three concepts as a loadable kernel module for Linux. Our working prototype demonstrates performance gains ranging from 5%-21% in Hadoop job execution time on 30 to 120-node Hadoop clusters.

1. Introduction

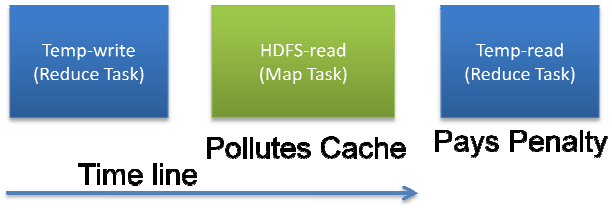

Figure-1

Consider a typical scenario in a Hadoop node when map and reduce tasks are running simultaneously. As fig. 1 shows there are 3 main I/O activities affecting the OS page cache. The first activity called Temp-write is where the reduce task writes to a temporary file during its shuffle/merge phase. Later this same temp file is read sequentially by the reducer (called the Temp-read activity in the third block in the figure.) Now in between these activities suppose a map task starts reading an HDFS file (called the HDFS-read activity in the fig. 1.) This HDFS-read activity pollutes the cache because the HDFS file will be never be read again in the future and is therefore not needed in the cache after it is read. But unfortunately due to the OS's LRU pagecache policy any free memory shortage situation causes temp file pages written during Temp-write to get ousted from the page cache to make space for the newly read HDFS file pages. Thus later during the Temp-read activity, these temp file pages have to be read from the disk instead of memory. In short the reducer's temp file read suffers a performance penalty due to the OS applying an LRU policy during the HDFS-read activity.

In the rest of paper we show how a combined LRU+MRU gives best performance for Hadoop applications compared to LRU alone.

2. Page Cache management

A key observation is that most files in Hadoop are only read-once and/or write-once. Hadoop applications access the following four types of files referred throughout the paper.

1. **Map HDFS files:** Maps read hdfs files, typically of size 128MB (DFS block size). Once read by mappers, it will be never used again in the near future.

2. **Map Temporary Files:** These files will be created and written to disk, and then all segments of the file will be read in a random order. All the contents of a file will be read exactly once. In abnormal conditions (in case of failures or errors) some segments will be re-read more than once. At the end of the job, the content of file will be deleted.
3. **Reduce Temporary File:** Temporary files will be created and written once, subsequently the file will be read sequentially once. At the end of job or task the contents of file will be deleted.
4. **Reduce HDFS file:** Reducers write files into HDFS that would be written once and will be never reused in the near term.

From the above classification, the files involved in hadoop are read-once and write once.

Prioritized EarlyWrite - Eviction - ReadAdvance

We have used a combination of three techniques called Prioritized Early Write, Eviction and Read Advance. These techniques are implemented along with MRU in a kernel module to improve the read and write throughput. Note that these techniques are not specific to hadoop, but apply wherever files are only read-once and/or write-once.

1.1. Prioritized EarlyWrite

In Linux, the dirty pages are flushed to disk only when the age of the page crosses a threshold (default is 30sec) (i.e. LRU policy) or the total amount of dirty pages crosses a threshold value (default is 10% of total memory).

Instead our basic idea is to convert this LRU writing to an MRU (Most Recently Used) i.e. flush the MRU pages that don't need to stay long in the cache, namely the HDFS file pages. The latter are not going to be accessed in the near future after being written to. However for temporary file dirty pages, we apply an LRU policy i.e. prolong their stay in the cache since they are going to be read by Hadoop after becoming dirty.

So for HDFS file dirty pages we flush them to disk as soon as the disk is idle instead of waiting for them to age or for the number of dirty pages to cross a threshold.

But for temp file dirty pages we prolong their cache stay by 1) writing them to disk only after all hdfs dirty pages are written out, and 2) increasing their age above the default 30 seconds.

The benefits of a longer stay of temp files in the cache are 1) the hadoop read operations become faster due to avoiding disk reads and 2) since temp files are short-lived i.e. since hadoop deletes them after it is done with the single round of read/write operation on them, we avoid the unnecessary writes of their dirty pages before they get deleted.

Additional Performance Gains due to EarlyWrite:

1. The average latency of a dirty page being flushed to the disk decreases since we avoid the delay in reaching the threshold of the dirty page limit or the page age.
2. The disk throughput increases as the writes are done when the disk is idle. That also indirectly provides priority to the reads.
3. Since we clean the dirty pages early, it helps Linux reclaim pages faster during memory shortage.

1.2. Eviction

With conventional Linux, such files will unnecessarily occupy the cache since the Linux page reclamation algorithm will not free them during memory shortage until they are aged enough. Instead during memory shortage, Linux may wrongly free many inactive temp-file pages just because the writes to them happened much earlier instead of recently read files. Later when hadoop has to read these same temp-file pages, it will have to read them from disk.

Instead our idea is to evict i.e. immediately free the pages that have been recently accessed but are not going to be accessed in the near future. In Hadoop the following pages fall in this evictable category: 1) pages of HDFS files read, usually by a map task 2) pages of HDFS files written by reducers, and 3) pages of temp files already read by their map or reduce task.

Performance Gains of Eviction:

1. The free pages created by eviction can be used for Read advance or Read-ahead (described in next section).
2. It will avoid freeing of useful but currently inactive pages (either map/reduce temp files or

the pages brought into cache by the read advance) during memory shortage.

Location where the eviction is done:

1. After reading a map HDFS file or writing a reduce HDFS the entire file is evacuated.
2. After reading a segment of a map temporary file, the segment alone is evacuated.
3. During reading of a reduce temporary file, the already read pages are evacuated.

1.3. Read Advance

In this optimization, pages are read in advance from the disk to cache in anticipation of the application's upcoming reads requests. It is a combination of a sequential read-ahead from the current read location and a random-access read of data from portions of the file which the application is anticipated to read from in the near future. This can significantly decrease the average read latency for a file. However if Read Advance is done during low free memory conditions it could worsen that condition. So we usually combine Read Advance with a corresponding Eviction (see previous section) of files not needed in the near future.

Differences between Read Advance and the conventional system-wide Read Ahead:

1. Read Advance reads the data from any offset of a file while Read Ahead only reads sequentially ahead of the current reading location.
2. Read Advance operates on a specific file while Read Ahead operates across all files in the system.
3. In Read Advance we can exploit our knowledge of the specific file's read patterns to read in a large number of pages into the cache in advance. This is generally not possible in Read Ahead since that operates across all files which makes it unsafe to read ahead a large number of pages.
4. When combined with our Eviction technique, the pages prefetched by Read Advance much less likely to be evicted in the near future as compared to using the conventional Read Ahead.

1.4. Combining All Three Techniques

Fig. 2. depicts the inter-dependency of the 3 techniques. The EarlyWrite creates a large number of clean pages (typically of HDFS files written to recently) which are then freed up by the Eviction technique thus making it possible for the Read Advance technique to read ahead a large number of pages from specific HDFS or temp file portions. Table 1 summarizes the optimization technique and the corresponding type of policy (LRU vs. MRU) applied to each type of Hadoop file read/write activity.

Type of file	**Page cache policy used**	**Techniques used**
HDFS file read	MRU	Read Advance + Eviction
HDFS file write	MRU	High priority Early Writes + Eviction
Map temp file write	LRU*	Low Priority Early Writes.
Map temp File Read	MRU	Read Advance + Eviction
Reduce temp file Write	LRU*	Low Priority Early Writes.
Reduce temp file Read	MRU	Read Advance + Eviction
** for our LRU policy we increase the page age above the default 30 sec.*		

Table-1 : FileTypes

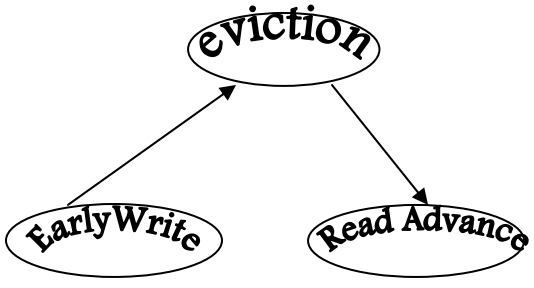

Figure-2 :Techniques in PageCache

The combined performance gains are:

1. Lower read call latency due to Read Advance and more temp file pages being in page cache.
2. Lower write call latency due to maintaining less dirty pages due to EarlyWrite.
3. Avoiding unnecessary writes of dirty temp file pages to disk when they get deleted shortly after being read.

3. Implementation details

We implemented all three techniques in a Linux kernel module (fig. 2.) consisting of three layers:

Initialization: It contains the initialize and close methods. The former alters the syscall table to use y_open instead of sys_open at module load time. The close method restores the original syscall table at module unload time..

System call layer: Upon an open system call, the y_open gets invoked which in turn calls sys_open. If the latter succeeds, y_open determines the file type based on the file prefix and read/write mode (table 2): if it is a hadoop file y_open stores the file handler along with file type (MRU/LRU) in local data structures. Let us call these as 'tagged' files.

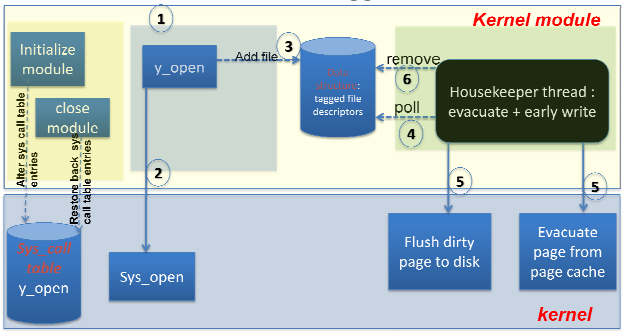

Figure-3 : Kernel Module Implementation

Housekeeping layer: This consists of housekeeping threads to periodically scan the tagged files for eviction and early write eligibility.

1. Prioritized Early write: There is one thread per physical disk device to flush the dirty pages.as per the MRU-LRU prioritization rules. Within the LRU priority level, map temp file pages are written first before reduce temp file pages.
2. Eviction: clears the pages of MRU files not needed anymore in the cache as explained previously.

Our current implementation is fully transparent to user-space since we automatically detect hadoop files based on their prefixes as shown in table 2. In the future we could add an ioctl interface for apps to pass requests to the kernel module containing the file type and r/w descriptor.

Type of File	File Prefix	File open mode as	type
Hdfs read	"blk_"	Read	MRU
Map temp write	"file."	Write	LRU
Map temp read	"file."	Read	MRU
Reduce temp read	"map_"	Read	MRU
Reduce temp write	"map_"	Write	LRU
Hdfs write	"blk_"	Read	MRU

Table-2 : Hadoop File Prefix

4. Test Results

In our tests conducted with single and multiple racks the maximum improvement was for a single rack with large reduce intensive jobs. Table-3 shows the improvements from various tests clearly showing that most of the runtime improvement comes from the I/O-intensive reduce phase.

Test setup (bench mark used; #nodes; #racks)	Avg. Map (optimized /baseline) seconds	Avg Reduce (optimized /baseline) seconds	Run time (optimized / baseline seconds)	Improvement in run time
GridMix v1; Nodes= 23; racks=1	17/17	145/185	5450 /6610	21.5%
GridMix V1; nodes= 53; racks=2	17/17	153/185	2540/2950	16 %

GridMix V3; nodes= 53; racks=2	13/13	44/46	16200 /17100	5%

Table-3 : Test Results

The average reduce time decreased from 185 sec to 145 sec for a single rack with a Gridmix-v1[1] Hadoop job load leading to 21.5% decrease in total runtime for the load. The second row of the table shows that the improvement for multi-rack is lesser at 16%; likely due to the network bottleneck between the racks. It is further lowered to 5% for a Gridmix-v3 job load since Gridmix-v3 loads are less I/O intensive than v1.

5. Future Work

Our page cache optimization can be improved further:

1. Instead of immediate eviction, increase the page age to a suitably high value so that it only gets evicted during low free memory situations.
2. Exploiting Read Advance better: During the shuffle phase we know in advance which random segments the task tracker needs to read from the map temp files to serve to a given reducer. So we use Read Advance to prefetch these random segments into the page cache.
3. Implementation improvements: It is much more efficient if our techniques are integrated into the kernel itself. We are currently prototyping a small footprint modular tiny kernel called 'Jiny'[2] which can incorporate different page cache implementations including those we wrote from scratch based on our MRU/LRU techniques. We use this prototype mainly to test our optimization on non-hadoop applications and to refine the model by fine tuning various parameters.

6. Conclusion

This paper described some key Hadoop-aware optimizations for the Linux kernel page cache that achieved significant Hadoop job runtime improvement of up to 21% depending on how I/O intensive the jobs were. These are fully transparent optimizations not needing any modification to Hadoop, Hadoop apps, or the Linux kernel.

References

[1] "Grid v1-v3 Mix," http://hadoop.apache.org/mapreduce/docs/current/gridmix.html.

[2] Naredula Janardhana Reddy, "Jiny Kernel", https://github.com/naredula-jana/Jiny-Kernel.