Adopting Two Strategies to Ensure and Optimize the Quality of Service in Linux

Shaohua Wan

Adopting Two Strategies to Ensure and Optimize the Quality of Service in Linux

Shaohua Wan
School of Information and Safety Engineering, Zhongnan University of Economics and Law, 430073 Wuhan, China
shwanhust@gmail.com

Abstract. This paper presents a new access-density-based prefetching strategy to improve prefetching for the access patterns, which have not been dealt with in the current Linux read-ahead algorithm. At the same time, motivated by the existing algorithms, we propose a hybrid and efficient replacement algorithm to improve buffer cache performance. Firstly, we propose the following three metrics to evaluate the above access patterns: reading file data backwards, reading files in a stride way (leaving holes between two adjacent references), alternating references between multiple file regions and reading files randomly. Secondly, having explored the eight representative recency/frequency-based buffer cache replacement algorithms, we carry on a hybrid replacement algorithm. Finally, these experimental results demonstrate the importance for buffer cache research to take both file system prefetching and replacement algorithm into consideration.

Results

Prefetching locates between page cache and disk buffer. According to [1], prefetching has significant impact on the performance of page cache replacement algorithms, while buffer cache replacement is critical to file system performance, so it worth to improve prefetching algorithm. The rationale of the design is that when pages in a region of a file are referenced, it is likely that pages around these pages may also be referenced. The entire file space is partitioned into a number of regions with the fixed size, and the number of pages which have been accessed, is tracked in a region. Once this number reaches a pre-determined threshold, prefetching makes all pages of the region resident in the buffer cache. So, further accesses of the pages in the same region can be hits. Moreover, to further overlap the computation time with I/O time, prefetching pages start in the adjacent regions when the number of accessed blocks in this region exceeds a higher pre-determined threshold. In this way, there can be no I/O stalls with the references to adjacent regions. This serves the same purpose as the operation of shifting two windows as references proceed in the Linux read-ahead. Only sequential access can be handled very well in a Linux kernel and can't handle random access and backward access. Figure 1(a) is sequential access, which can be handled very well in a Linux kernel. That is why there are so many fewer misses and so many more hits. Figure 1(b), however, is random access, so it shows no hits and all misses. As can be seen in this figure, we can also notice that how far the curve that represents the number of misses blocks from X axis. the curve that represents "number of misses and number of prefetched" is very close to the curve that represents "number of hits and number of misses" in Fig. 1(a). Since the number of misses is very less (almost zero), the prefetched blocks are being used by user's requests, this means that

the precision is good and the prefetching policy is performing very well. In Fig. 1(b) The curve that represents “number of misses and number of prefetched” is overlapping with the curve that represents “number of hits and number of misses”, the reason for this can be explained by the fact that number of misses is used in both curves and the values of number of prefetched and hits are both zero. Again, the number of prefetched blocks is zero, so the prefetching policy in Fig. 1(b) is performing very badly. As for “Number of Prefetched but not yet Requested Blocks” metric, in Fig. 1 (a), almost all blocks that are prefetched are being used and this prefetching policy can guarantee cache buffer to have more spaces at its disposal for a higher hit ratio. On the contrary, the number of prefetched blocks is zero and the cache space does not function in Fig. 1(b). Therefore, from Fig. 1 (a) and (b), we draw a conclusion that the current Linux kernel cannot handle non-sequential workload access pattern.

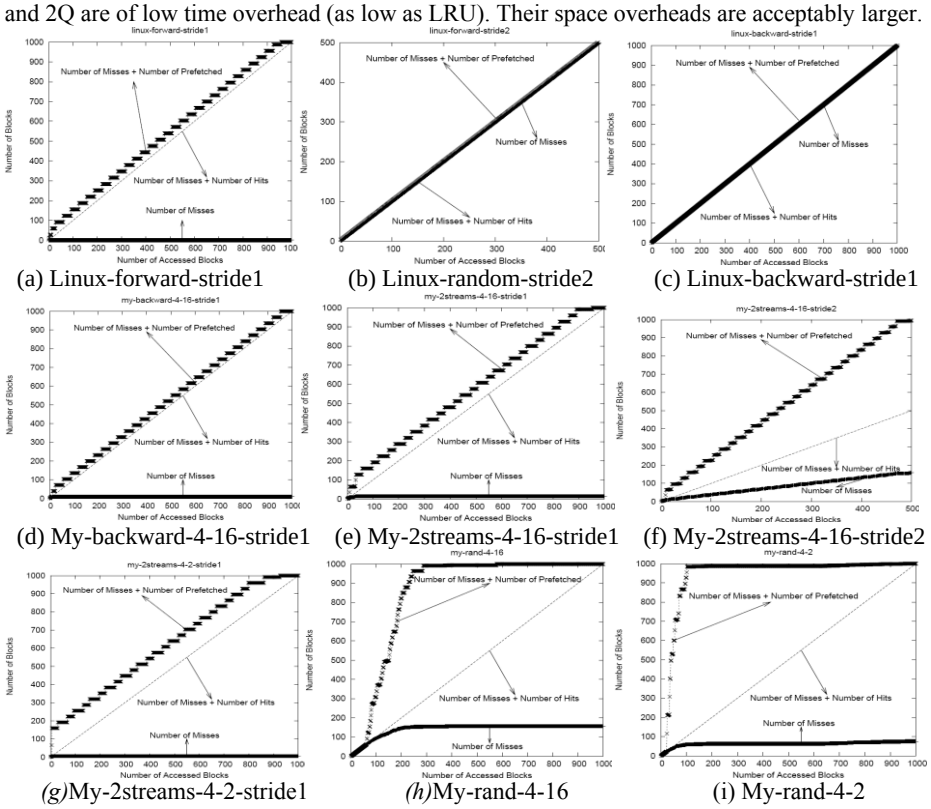
As for backward access patterns, Linux kernel also can’t handle it. So what is shown in the “linux-backward-stride1” curve, Figure 1(c) is all misses and no hits or prefetching. As for Figure 1(d), because of the region prefetching algorithm, all the blocks have been prefetched before being used. So it shows much more hits and fewer misses than Figure 1(c), which indicates the “My-backward-4-16-stride1” prefetching policy has much more accuracy and precision than Linux kernel. The comparison of space overheads in these two curves is similar to that in Fig. 1(a) and Fig. 1 (b).

For Figures 1(e) and 1(f), both two curves that represent the number of misses are approximately close to X axis, which shows there are extremely fewer misses and those two prefetching policies are able to predict user’s future requests. In this case, we can say that both are equally accurate. But the upper line doesn’t completely overlap the middle one. It means there are a few blocks prefetched which are useless. Figure 1(f) indicates the inability of the regional algorithm to handle the access pattern perfectly. That is why the number of missing blocks increases and the gap between the upper line and the middle line enlarges. It also means that only a small fraction of prefetched blocks are actually used later. In consequence, we can say that the “My-2streams-4-16-stride1” prefetching policy is much more precise than “My-2streams-4-16-stride2” policy. We also notice that almost all prefetched blocks are the same as the accessed blocks, in which the space overhead is low. While in Figure 1(f) we see that not all prefetched blocks are accessed, which means that the space overhead will be increased. The space overhead of the “My-2streams-4-16-stride1” prefetching policy is lower than the space overhead of the “My-2streams-4-16-stride2” policy.

Intuitively, there is no obvious difference between Figure 1(e) and Figure 1(g). The only difference of the algorithm in these two figures is the value of the high watermark that triggers the prefetching of pages in its adjacent regions and the low watermarks are same. However, zooming in on the two figures reveals the “.cuv” file directly; Figure 1(g) prefetches more aggressively than Figure 1(e). This can be demonstrated by the fact that the upper line is closer to the middle line in Figure 1(e). The fact that the gap between the upper line and the middle line becomes larger in Figure 1(g) shows that only a small fraction of prefetched blocks are actually used later. Thereby, the prefetching policy of “My-2streams-4-16-stride1” outperforms that of “My-2streams-4-2-stride1” in terms of the precision metric. We can also observe that the space overheads will be decreasing in both prefetching policies.

As for Figure 1(h) and Figure 1(i), the only difference is the size of the expand threshold. As can be seen in these figures, the curves that represent the number of misses are not overlapping with X axis, which indicates there are a few prefetched blocks that are not used. In terms of the accuracy metric, both prefetching policies don’t perform perfectly. Because Figure 1(i) uses more aggressively prefetching, there are fewer misses at the cost of less precise prefetching. That is why the upper line in Figure 1(i) is further away from the middle one than that of Figure 1(h). Finally, both graphs show that, the whole file is prefetched in the beginning and all the blocks are occupying the cache space and need to wait sometime before they are requested by user’s program. This means that the space overhead will be increasing and both prefetching policies are performing very badly in terms of space overhead.

We discuss the eight representative recency/frequency-based buffer cache replacement algorithms (OPT, LRU, LRFU, LRU-K, LIRS, 2Q, ARC, Hybrid) used in our evaluation of hit ratio,. For each replacement algorithm, we summarize the original algorithm followed by the adapted version that manages the blocks brought in by kernel. The motivation is simply to compare the different algorithms in a realistic scenario when implemented in the Linux buffer cache. A hybrid replacement algorithm is just that, when the cache is full it randomly implements the above-mentioned one replacement algorithm and picks which page will be replaced. In Figure 2 (a), this shows the hit ratios of the hybrid replacement algorithm and other algorithms mentioned before. Here I also include the OPT replacement, an off-line optimal replacement algorithm, which depends on the knowledge about the future accesses for its decision. In this hit ratio graph, X axis is for cache size, Y axis is for hit ratio. The hybrid hit ratio is very close to the optimal, much better than others. We can see that LRU hit ratios are very low before the cache reaches 400 blocks, the size of one of its locality scopes. LRU is not effective until this working set fully reside in the cache. The hybrid algorithm achieves as high hit ratio as LIRS algorithm in Figure 2(b). Figure 2(c) shows the easy case for all the replacement algorithms. All the curves are close to that of LRU. They all have high hit ratios. LIRS uses both IRR (or reuse distance) and recency for its replacement decision while 2Q uses only reuse distance. LIRS adapts to the locality changes when deciding which blocks have small IRRs. 2Q uses a fixed threshold in looking for blocks of small reuse distances. Both LIRS and 2Q are of low time overhead (as low as LRU). Their space overheads are acceptably larger.

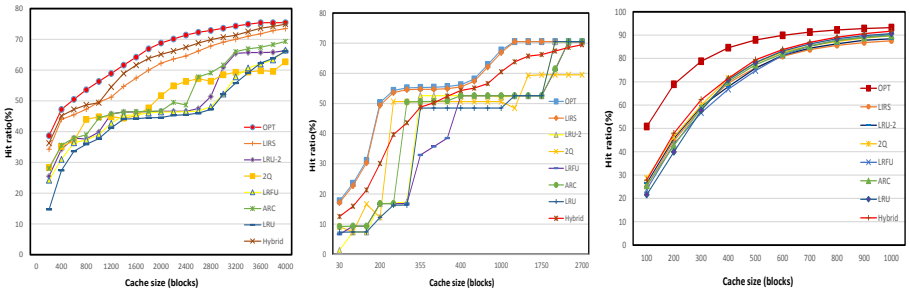


(a) Linux-forward-stride1 (b) Linux-random-stride2 (c) Linux-backward-stride1

(d) My-backward-4-16-stride1 (e) My-2streams-4-16-stride1 (f) My-2streams-4-16-stride2

*(g)*My-2streams-4-2-stride1 *(h)*My-rand-4-16 (i) My-rand-4-2

Figure 1. The various curves of performance characteristics of the two prefetching policies



(a) (b) (c)

Figure 2. Hit ratio curves by various replacement policies on various workloads: postgres, multi2 and sprite.

Acknowledgments. We would like to thank the anonymous reviewers for their insight and suggestions which have substantially improved the content and presentation of this paper. This work was supported by the Fundamental Research Funds for the Central Universities of China under Grant No. 31141311303 and the Research Project Funds (32114113001).

References

1. Ali R.Butt,Chris Gniady, and Y. Charlie Hu: The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms. Proceedings of the ACM International Conference on Measurement & Modeling of Computer Systems (SIGMETRICS), Banff, Canada, June 6-10, 2001.