LETTER

Application Prefetcher Design Using both I/O Reordering and I/O Interleaving*

Yongsoo JOO[†], ***Member***, **Sangsoo PARK**[††a)], ***and*** **Hyokyung BAHN**[††], ***Nonmembers***

SUMMARY Application prefetchers improve application launch performance on HDDs through either I/O reordering or I/O interleaving, but there has been no proposal to combine the two techniques. We present a new algorithm to combine both approaches, and demonstrate that it reduces cold start launch time by 50%.

key words: *I/O reordering, I/O interleaving, application prefetching*

1. Introduction

Modern operating systems (OSes) use a demand paging system to efficiently manage limited main memory shared by user applications. However, it does not perform well when launching an application in a "cold start" situation (i.e., none of its launch data blocks are hit in the OS page cache). In particular, it typically generates a non-sequential read accesses from hundreds of files on a hard disk drive (HDD), leading to poor disk throughput. Furthermore, a CPU is frequently blocked while waiting for disk I/O completion.

This launch inefficiency can be largely mitigated by adopting solid state drives (SSDs), but there are still many systems that cannot afford the high cost per bit of SSDs, such as set-top boxes, video game consoles, and low-cost laptop or desktop PCs. Hence, optimization techniques that improve application launch performance on HDDs will remain useful until HDDs totally disappear from the market.

Application prefetching is a representative technique to mitigate the launch inefficiency. Once an application accesses a set of data blocks in a particular order during a launch, it is highly likely to access almost the same set of data blocks in almost the same order in the next launches [1]–[3]. Such an I/O sequence is called an application launch sequence in recent work [2]. The deterministic launch behavior of applications allows application prefetchers to predict with high accuracy what data block will be requested next. I/O Reordering and I/O interleaving are the two key techniques behind the application prefetchers.

Suppose an application launch sequence consists of 10 I/O requests, and their logical block addresses (LBAs) are $(10, 5, 9, 4, 8, 3, 7, 2, 6, 1)$ in the order of occurrence. Figure 1 (a) depicts the activity of a CPU and a HDD under a cold start scenario with no prefetching, where each data block is fetched from the HDD on demand, showing the inefficiency of the demand paging discussed above.

To mitigate the launch inefficiency, an "I/O reordering" technique (Fig. 1 (b)) can be used [3]. First, a prefetcher obtains an application launch sequence for each application by monitoring what data blocks are fetched during launch time. When the next launch is detected, it pauses the application, reorders and fetches the launch data blocks specified in the application launch sequence all at once, and then resumes the application. In this way, it can effectively reduce disk seek time as well as allow the application to launch without experiencing a page fault.

FAST [2]—originally proposed for SSDs—is an alternative approach by overlapping CPU computation with I/O accesses. This sort of "I/O interleaving" technique has been studied for HDDs as well [4]. Figure 1 (c) shows that the prefetcher runs in parallel with the application, fetching the application launch sequence in its original order.

According to our best knowledge, there has been no proposal to combine the two techniques, and thus we propose a new application prefetcher to take advantage of the both. Its key idea is to segment an application launch sequence into multiple non-overlapping and contiguous subsequences and reorder each subsequence in LBA order.

Figure 1 (d) shows our approach, where the example sequence is split and sorted into $(4, 5, 9, 10)$, $(3, 7, 8)$, and $(1, 2, 6)$. Once $(4, 5, 9, 10)$ has been fetched, the applica-

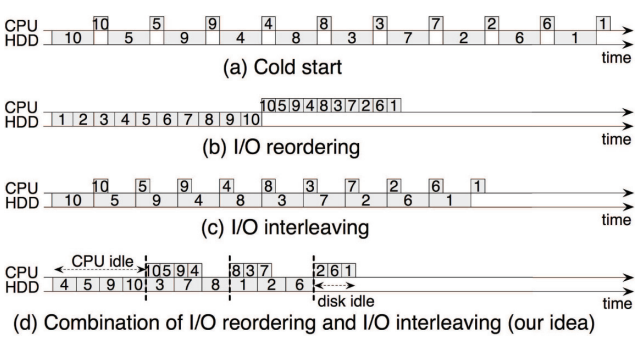

Fig. 1 CPU and disk usage of different launch methods. The number in each box denotes an LBA, with the box width representing an average processing time (not in scale).

Manuscript received June 1, 2015.
Manuscript revised July 27, 2015.
Manuscript publicized August 20, 2015.

[†]The author is with the School of Computer Science, Kookmin Univ., Korea.
[††]The authors are with the Dept. CSE, Ewha Womans Univ., Korea.

*This research was supported by the National Research Foundation of Korea Grant funded by the Korean Government (NRF-2014S1A5B6037290) and MSIP (2011-0028825).

a) E-mail: sangsoo.park@ewha.ac.kr (Corresponding author)

DOI: 10.1587/transinf.2015EDL8125

tion uses it to begin the launch process, while the prefetcher fetches $(3, 7, 8)$ in parallel. Likewise, the CPU computation for $(3, 7, 8)$ can be performed simultaneously with fetching $(1, 2, 6)$. Hence, CPU computation can be overlapped with disk accesses, while still benefiting from I/O reordering.

As finding an optimal segmentation is the key for our approach, we made the followiing contributions in this work. First, we expressed the problem of finding an optimal prefetching schedule as a sequence segmentation problem, and suggested its suboptimal version to make it more tractable. Second, we proposed a novel algorithm that can quickly find a solution to the suboptimal problem. Finally, we demonstrated that our method could increase both CPU and disk utilization while effectively optimizing disk access order, outperforming the state-of-the-art application prefetching techniques.

2. Finding Optimal Segmentation

If we try a brute-force search to find the optimal segmentation for an application launch sequence consisting of n I/O requests, we need to investigate $2^{(n-1)}$ cases, as there are $n-1$ possible breakpoints between the I/O requests. Since n ranges from hundreds to thousands for typical applications, it is not possible to examine all cases, and thus we decide to take a heuristic approach.

To begin with, we make a set of observations. First, a CPU should remain idle until fetching the first subsequence, as shown in Fig. 1 (d), which may be regarded as a pre-buffering phase. Likewise, a HDD is idle while the CPU computation is in progress for the last subsequence, as there is no more data block to be fetched. Second, an application launch process is typically I/O bound on HDDs, meaning that the time to fetch launch data from the HDD is greater than the total CPU time. This implies that if the pre-buffer size is too small, CPU usage will drop significantly due to excessive page faults. On the other hand, if it is too large, disk usage will drop to 0% before launch completion (e.g., Fig. 1 (b) shows an extreme case).

These observations lead us to speculate that a desirable segmentation would have a balanced pre-buffer size such that: (1) the CPU will maintain 100% utilization except for the pre-buffering phase; (2) the HDD will maintain 100% utilization except for the CPU time for the last subsequence; and (3) the size of the last subsequence would desirably be as small as possible to maximize disk utilization during launch time. Based on the speculation, we formulate a suboptimal version of the original problem, expecting that it could significantly reduce a solution space without much loss of optimality.

Problem 1: Given an application launch sequence $P = (p_1, \ldots, p_n)$, find a segmentation that splits P into multiple non-overlapping and continuous subsequences such that after fetching the first subsequence, CPU and HDD usage stays at 100% except for the computation of p_n.

Here p_i is a three-tuple of values (starting LBA a_i, size s_i, CPU time c_i).

3. Proposed Segmentation Method

We propose a novel method that can quickly find the solution to Problem 1 while not degrading the solution quality. The key idea is to traverse an application launch sequence in a reverse direction, taking subsequences one by one from the tail while satisfying the CPU and disk usage constraints of Problem 1.

Algorithm 1 describes how P is segmented into m subsequences $Q = (q_m, q_{m-1}, \ldots, q_1)$. A subsequence q_i is expressed as $q_i = (p_{k_i+1}, \ldots, p_{k_{i-1}})$, where $k_i < k_{i-1}$, $k_m = 0$, and $k_0 = n$. We use Figs. 2 (a)–(e) to explain how Algorithm 1 splits $(p_1, \ldots, p_{24})$ into five subsequences. First, in Fig. 2 (a), a seed subsequence q_1 is set to (p_{24}) according to the constraint of Problem 1. Next, in Fig. 2 (b), DISK_ACCESS() is called to measure the time t_1 to fetch p_{24} from a HDD. The CPU time is then accumulated from c_{23} to c_{21} until it equals or exceeds t_1, and thereby the next subsequence q_2 is determined to be (p_{21}, p_{22}, p_{23}). In Fig. 2 (c), $\hat{q}_2 = (p_{21}, p_{23}, p_{22})$, which is the sorted version of q_2, is fed into DISK_ACCESS() to obtain t_2, which is used again to determine $q_3 = (p_{15}, \ldots, p_{20})$. The same procedure is repeated to obtain $q_4 = (p_5, \ldots, p_{14})$, and $q_5 = (p_1, \ldots, p_4)$, as shown in Figs. 2 (d) and (e). Note that in Fig. 2 (d), the value of $\sum_{i=5}^{14} c_i$ does not exactly match t_3, causing a disk usage drop. Other than the drop, the CPU and disk utilization are maintained 100% in the time periods t_1, t_2, and t_3, but not for t_4, which is unavoidable because there is no more I/O request to fill t_4.

Line 12 of Algorithm 1 calculates the expected launch time t_{exp} with the obtained subsequences. The first two terms are the time to fetch the first two subsequences, and the third term is the accumulated CPU time except for the first subsequence (e.g., $t_5 + t_4 + \sum_{i=5}^{24} c_i$ in Fig. 2 (f)).

Algorithm 1 Sequence segmentation algorithm

```
1:  procedure SEGMENT(P = (p_1, ..., p_n))
2:      i = 0; k_1 = n − 1; q_1 = (p_n)
3:      repeat
4:          i = i + 1
5:          q̂_i = SORT(q_i)    // sort q_i in LBA order
6:          t_i = DISK_ACCESS(q̂_i)    // t_i: the measured time to fetch q̂_i
    from a disk
7:          k_{i+1} = GET_NEXT_SUBSEQENCE(k_i, t_i)
8:          q_{i+1} = (p_{k_{i+1}+1}, ..., p_{k_i})
9:      until k_{i+1} = 0    // i.e., q_{i+1} = (p_1, ..., p_{k_i})
10:     m = i + 1    // m: the total number of subsequences
11:     q̂_m = SORT(q_m); t_m = DISK_ACCESS(q̂_m)
12:     t_exp = t_m + t_{m−1} + Σ_{j=(k_{m−1}+1)}^{n} c_j
13:     return Q = (q_m, ..., q_1) and t_exp
14: end procedure
15: function GET_NEXT_SUBSEQUENCE(k_i, t_i)
16:     k_{i+1} = k_i
17:     repeat
18:         k_{i+1} = k_{i+1} − 1
19:     until Σ_{j=k_{i+1}+1}^{k_i} c_j ≥ t_i or k_{i+1} = 0
20:     return k_{i+1}
21: end function
```

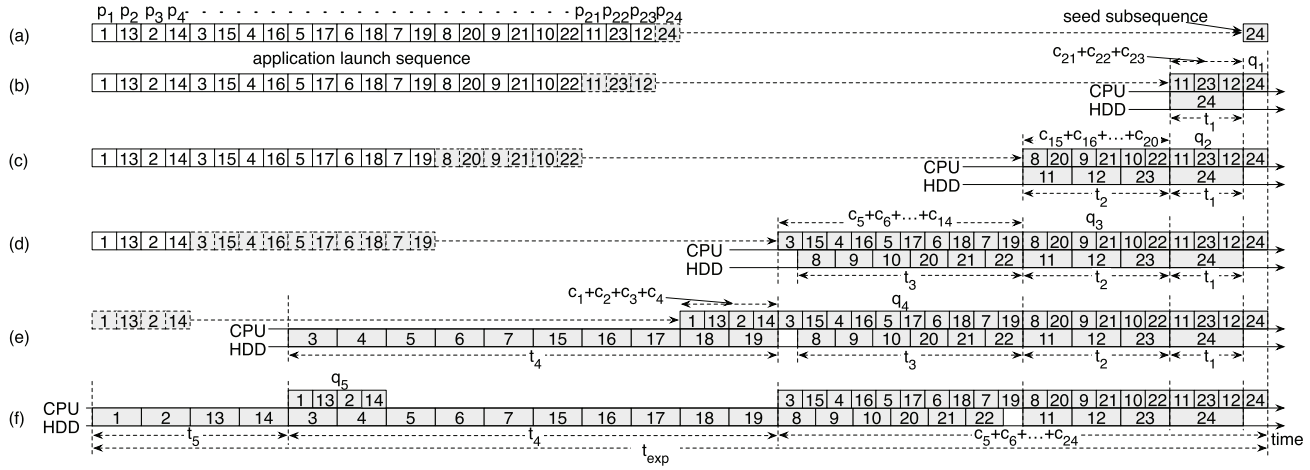

Fig. 2 Segmenting $P = (p_1, \ldots, p_{24})$ into $Q = (q_5, \ldots, q_1)$. (a) A seed subsequence q_1 is taken from P. (b)–(e) Algorithm 1 takes four iterations to obtain q_2, q_3, q_4, and q_5. (f) Calculation of t_{exp}.

The running time of Algorithm 1 is mostly determined by SORT(), GET_NEXT_SUBSEQUENCE(), and DISK_ACCESS(). The complexity of the first two functions are $O(n \log n)$ and $O(n)$, respectively, and their input size n typically does not exceed 2000 for most applications. Their execution time is thus negligible compared to that of DISK_ACCESS() that makes access to a HDD. When running Algorithm 1, DISK_ACCESS() is called m times, once for each of m subsequences with each subsequence sorted by LBA. Consequently, the execution time of Algorithm 1 becomes close to t_{exp}.

4. Performance Evaluation

Emulation platform. For rapid prototyping of application prefetchers, we developed an emulation platform running on the Linux OS. The platform supports flexible priority adjustment between applications and prefetchers, enabling convenient switching between prefetching methods. It allows application prefetchers to bypass the file system and the block layer of the host OS, having a full control of issuing block I/O requests as they intend. To support this, a target HDD is opened as a block device using `open()` with "`O_DIRECT`" flag. It emulates all kernel-level I/O optimizations such as buffered I/O, I/O merging, and I/O reordering that are performed by the Linux block layer functions, which is achievable by profiling application launch sequences between the Linux block layer and the device driver.

The platform runs on a real HDD to provide accurate and realistic measurements of I/O latency, which is in contrast to disk simulation tools [5] that rely on an analytic performance model of HDDs. It reports application launch time by monitoring when the CPU computation for c_n is finished. Although the platform does not support NCQ (native command queueing), it does not degrade much the accuracy of evaluating application launch performance where blocking read requests dominate (i.e., effective queue depth seldom exceeds 2 or more).

Experimental setup. We installed Ubuntu 12.04.2 LTS with an EXT4 file system on a clean Intel machine equipped with i5-3470 CPU, 16 GB of main memory, and a 2 TB 7200 RPM HDD (WD2002FAEX). We then installed 12 Linux applications, and captured their I/O sequences during launch time using `blktrace` when the machine is idle (i.e., not running other user applications). We trimmed the trace assuming a launch process completes if the HDD is idle for 1.0s or more. We then post-processed the obtained traces as explained in Sect. 3 to get application launch sequences.

Launch time measurement. We captured 11 different application launch sequences for each test application, using one for the prefetcher input and the others for emulating its 10 different launches. We measured application launch time for four different launch methods: (1) a cold start, (2) I/O interleaving, (3) I/O reordering, and (4) the proposed method. Figure 3 shows the average launch time of the ten individual launches for each application, together with the average prefetcher miss rate. The proposed method reduces launch time by 39% to 66%, outperforming the I/O reordering method for every application. In particular, it takes 63.5s to launch all applications, corresponding to a 22% improvement from the I/O reordering method.

CPU and disk usage profiling. To observe how the baseline methods and the proposed prefetcher behave differently during launch time, we monitored CPU and disk usage along with LBAs. Figure 4 (a) shows the typical behavior of the demand paging system mentioned in Sect. 1: (1) a non-sequential disk access pattern is observed for the entire launch time period; and (2) when the disk usage is high, the CPU usage becomes low. Figures 4 (b) and (c) also show that the two conventional prefetching techniques perform as expected in accordance with Fig 1.

Figure 4 (d) shows the behavior of the proposed prefetcher, where the application launch sequence is split into 6 subsequences, of which the sizes are 672, 1919, 550, 463, 30, and 1. The CPU and disk usage remains high during the time interval of 3.3s to 5.6s (i.e., fetching the third to

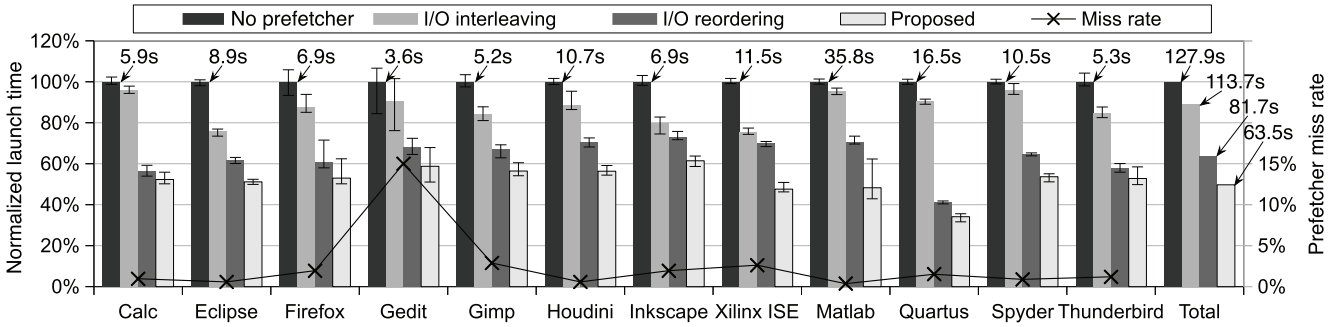

Fig. 3 Launch performance of different prefetching methods (the left y-axis is normalized to cold start time, and the error bars represent the minimum and maximum values).

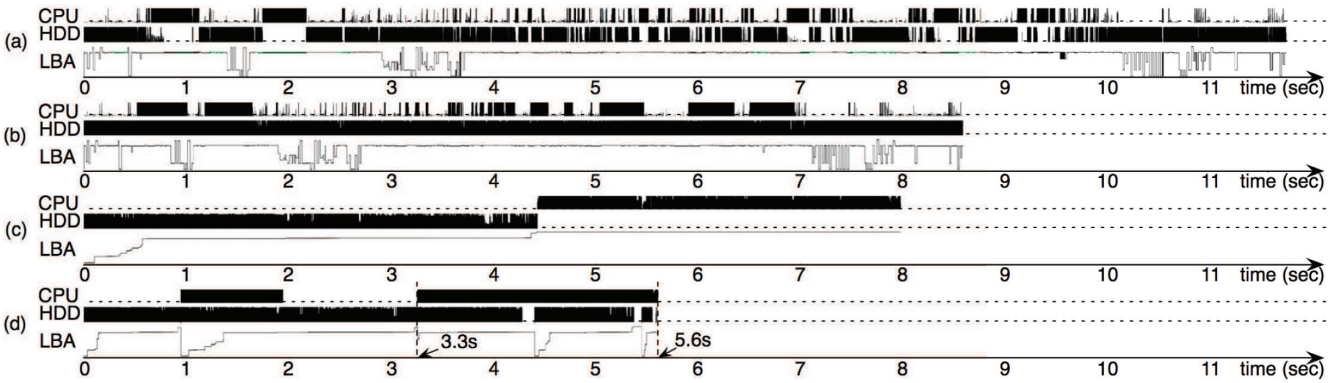

Fig. 4 CPU and disk usage with normalized LBAs (application: Xilinx ISE, sampling period: 1 ms). (a) Cold start. (b) I/O interleaving. (c) I/O reordering. (d) Proposed.

the last subsequences), but there appear a small amount of disk idle periods.

Estimation accuracy. We obtained application launch sequences, one for each of 12 applications, and ran Algorithm 1 to get segmented launch sequences with their expected launch time t_{exp}. We then ran the proposed prefetcher with the segmented sequences to compare the resulting launch time with t_{exp}. The reported estimation error of Algorithm 1 ranges from −3% to 29% (7% on average).

Time and space overhead. The total execution time of Algorithm 1 for all 12 applications was 50.7s, while the sum of t_{exp} values calculated by Algorithm 1 was 54.0s, which coincides with the discussion of Sect. 3. When the algorithm is deployed in a real system, this overhead can be effectively hidden by running it when the system is idle. The total size of the application launch sequences and their segmented versions for 12 applications were 1.9 MB, which can be further reduced by using a binary format instead of a text format.

5. Conclusion and Future Work

We presented a novel application prefetching technique that can exploit both I/O reordering and I/O interleaving, and demonstrated that it outperforms the state-of-the-art application prefetchers. As our method performs analysis on the fly on a real disk rather than relying on a pre-analyzed disk performance model, it is immediately applicable to any model of HDD. Our future work includes the following. First, we will combine our approach with disk layout modification methods [1], [6], which can effectively overcome the poor I/O performance due to a non-sequential disk access pattern. Second, although we used LBA as the sort key for the I/O reordering method, which is shown to be effective in many literatures [7], [8], we may further optimize application launch performance by using more sophisticated disk scheduling policies, such as SSTF [8].

References

[1] M. Bhadkamkar, J. Guerra, L. Useche, S. Burnett, J. Liptak, R. Rangaswami, and V. Hristidis, "BORG: Block-reORGanization for self-optimizing storage systems," Proc. FAST, pp.183–196, 2009.

[2] Y. Joo, J. Ryu, S. Park, and K.G. Shin, "FAST: Quick application launch on solid-state drives," Proc. FAST, pp.259–272, 2011.

[3] M.E. Russinovich and D. Solomon, Microsoft Windows Internals, 4th ed., pp.458–462, Microsoft Press, 2004.

[4] Z. Li, Z. Chen, S.M. Srinivasan, and Y. Zhou, "C-Miner: Mining block correlations in storage systems," Proc. FAST, pp.173–186, 2004.

[5] J.S. Bucy and G.R. Ganger, "The DiskSim simulation environment version 3.0 reference manual," Tech. Rep. CMU-CS-03-102, Department of Computer Science, Carnegie-Mellon University, Jan. 2003.

[6] H. Huang, W. Hung, and K.G. Shin, "FS2: Dynamic data replication in free disk space for improving disk performance and energy consumption," Proc. Twentieth ACM Symposium on Operating Systems Principles, SOSP '05, pp.263–276, 2005.

[7] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang, "DiskSeen: Ex-

ploiting disk layout and access history to enhance I/O prefetch," Proc. USENIX ATC, pp.261–274, 2007.

[8] B.L. Worthington, G.R. Ganger, and Y.N. Patt, "Scheduling algorithms for modern disk drives," Proc. ACM SIGMETRICS Performance Evaluation Review, vol.22, no.1, pp.241–251, 1994.