Exploiting SSD parallelism to accelerate application launch on SSDs

J. Ryu, Y. Joo, S. Park, H. Shin and K.G. Shin

Using an optimised application prefetcher to accelerate the application launch on solid-state drives (SSDs) by exploiting the SSD parallelism is proposed. The proposed prefetcher was implemented on the Linux OS and achieved a 37% reduction of prefetcher execution time, which corresponds to an 18% reduction of application launch time.

Introduction: Flash-based solid-state drives (SSDs) typically consist of multiple NAND flash chips. The I/O performance of a SSD strongly depends on how well the SSD controller exploits its internal parallelism for different I/O patterns. For example, a SSD controller can easily activate multiple flash chips at the same time for sequential I/O requests, a large size I/O request, and multiple concurrent read requests. For another example, recent work [1] reported that the SSD parallelism is not utilised well if write requests are mixed with read requests, and proposed a new SSD scheduler to avoid such inefficiency.

There has not been enough analysis on how the SSD parallelism is utilised for the application launch on PCs. An application typically generates thousands of random read requests during its launch, resulting in excessive disk head movements of a hard disk drive (HDD). As SSDs do not incur delays owing to the disk head positioning time, the application launch performance can be significantly improved by replacing a HDD with a SSD. However, traditional HDD-aware optimisation schemes that optimise disk head movements are not recommended for use for a SSD as it has no disk head. For example, Windows 7 disables its application prefetcher and disk defragmenter upon detection of a SSD [2]. Unfortunately, such a strategy prevents exploiting the SSD parallelism for an application launch. We observed that only one outstanding request is sent to the SSD during the most time periods of an application launch.

In this Letter, we demonstrate that using an application prefetcher can help exploit the SSD parallelism during application launch time. We then propose the two-phase application prefetcher to improve application launch performance further by maximising the effective number of outstanding I/O requests. We implemented the proposed application prefetcher on the Linux OS, and demonstrated a 37% reduction of prefetcher execution time, which corresponds to an 18% reduction of application launch time.

SSD structure: Most modern flash-based SSDs consist of multiple NAND flash chips. A well-designed SSD controller can utilise the internal parallelism of the SSD to increase the SSD performance beyond that of a single flash chip. SSDs are usually connected to a host PC through a serial advanced technology attachment (SATA) II interface, which supports native command queueing (NCQ). Using the NCQ feature, the host PC can send up to 32 outstanding I/O requests to the SSD controller, allowing SSDs to process them in parallel.

Application prefetcher: An application prefetcher (e.g. Windows prefetcher [3]) improves application launch performance by optimising disk head movements during application launch time. The set of block requests generated during application launch time changes little over repeated launches, and thus the application prefetcher can extract such a set by monitoring block requests to the HDD. Upon detection of the launch of a target application, the application prefetcher works as follows: 1. it immediately pauses the execution of the target application; 2. it fetches the predetermined set of block requests from the HDD according to the sorted order of their logical block addresses (LBAs); and 3. then it resumes the target application.

Using application prefetcher on SSDs: Fig. 1*a* shows a typical application launch process on a SSD, where d_i is a data block request issued during the launch process. After d_i is fetched, the CPU continues its computation, which is denoted by c_i, until the next page fault occurs. Queue depth means the number of block requests being processed in the SSD at a given time. In this example, we assumed that each data block request has the following dependency with its associated metadata block request m_j: 1. m_1 should be fetched prior to d_1, d_2, and d_3; and 2. m_4 should be fetched prior to d_4 and d_5. We also assumed that the LBA order of the data blocks is given as: $d_4 < d_5 < d_2 < d_1 < d_3$.

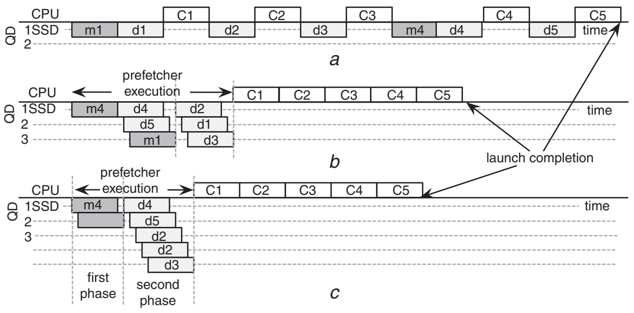

Fig. 1 *Application launch procedure (x-axis not in scale)*

a No prefetcher
b Baseline prefetcher
c Two-phase prefetcher
QD = queue depth

In Fig. 1*a*, the queue depth is not increased more than 1, which is because there is no way for the application to know what to fetch next until a page fault occurs. This is the inherent limitation of the demand paging, i.e. all the blocks required for the application launch are fetched on demand. To overcome this inefficiency, we suggest using an application prefetcher. As the application prefetcher knows all the blocks required for the launch, it can continuously send block requests to the SSD without waiting for the next page fault. Hence, the effective queue depth can be increased beyond 1, as shown in Fig. 1*b*.

Two-phase application prefetcher: Even if we use an application prefetcher, there still exists dependency between metadata and normal data, preventing the queue depth to reach the maximum value of 32. Fig. 1*b* shows that issuing d_4 and d_2 are blocked waiting for the completion of m_4 and m_1, respectively. To resolve this problem, we propose the two-phase application prefetcher. In the first phase, we gather all the block requests for metadata and issue them first. In the second phase, we issue the remained block requests, which are for normal data. As the block requests in each phase have no dependency among them, they can be continuously issued without being blocked, as shown in Fig. 1*c*.

We implemented the two-phase application prefetcher on the EXT3 file system, which is widely used with the Linux OS. As there is no system call provided by the Linux OS for explicitly fetching metadata (e.g. inode blocks, indirect pointer blocks) at user level, we implemented the proposed prefetcher as a kernel module to use the kernel function ll_rw_block() with the READA argument. For the normal data block of regular files, we used force_page_cache_readahead() to fetch them asynchronously.

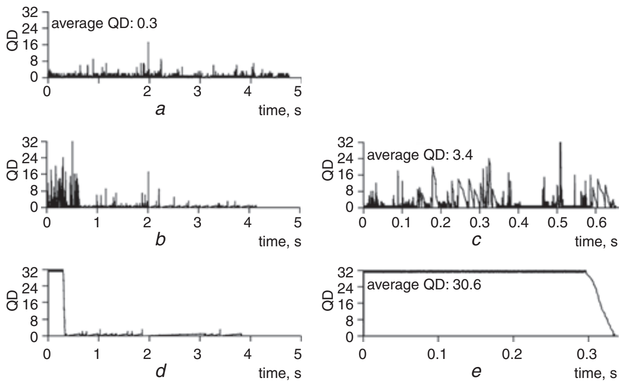

Fig. 2 *Visualisation of SSD queue depth (application: Eclipse)*

a No prefetcher
b Baseline prefetcher
c Baseline prefetcher (zoomed in)
d Two-phase prefetcher
e Two-phase prefetcher (zoomed in)
QD = queue depth

Results: We performed an experiment on a PC equipped with an Intel i7-860 2.8 GHz CPU and an Intel 80 GB SSD (X25-M G2, 10 channels), where we installed a Fedora 12 with Linux kernel 2.6.35. As we are interested in the application launch performance in a cold start scenario, i.e. all the data needed for the launch are not in the main memory page cache, we flushed the page cache using the following command: sync; echo 3>/proc/sys/vm/drop_caches. For comparison purpose, we also measured application launch time in a warm start scenario, i.e. all the requested data are found in the page cache, by executing the prefetcher first, and then launching the application.

Fig. 2 depicts the queue depth for each application launch scenario, where the dashed boxes are the time periods the application prefetchers are executed. Fig. 2*a* shows that the average queue depth is less than 1, which is as expected. There are few points where the queue depth is larger than 1, which is because the application created multiple threads. Figs. 2*b* and *c* show that the application prefetcher increased the queue depth beyond 1. However, the achieved queue depth of 3.4 is still far smaller than 10, the number of channels of the SSD we used. Figs. 2*d* and *e* show that the two-phase application prefetcher successfully increased the queue depth to the maximum value of 32.

We measured the prefetcher execution time and application launch time for various Linux applications. Fig. 3*a* shows that the two-phase application prefetcher reduced the prefetcher execution time by 37% compared with the baseline prefetcher. As a result, the two-phase application prefetcher reduced application launch time by 18% compared with the cold start launch time, as shown in Fig. 3*b*.

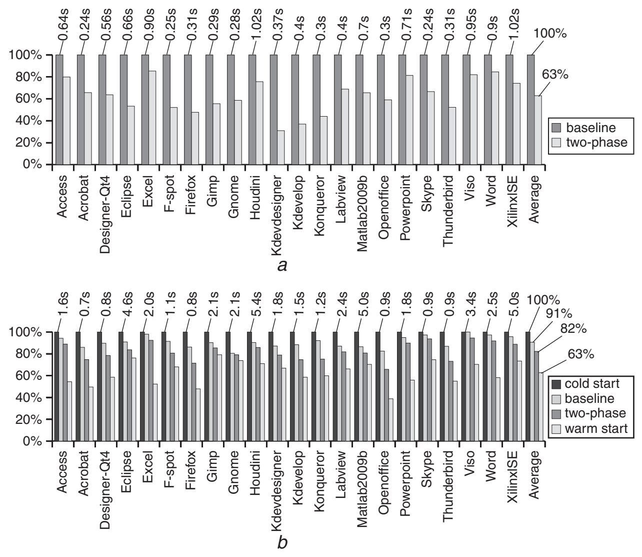

Fig. 3 *Measured prefetcher execution time and application launch time*

a Prefetcher execution time
b Application launch time

Conclusion and future work: We have demonstrated how the SSD parallelism can be utilised to improve application launch performance. Experimental results show that the proposed two-phase application prefetcher reduces application launch time by 18%, which is an immediate benefit because existing tools such as the Windows prefetcher [3] can be used with a slight modification. Also, we recently suggested another application launch performance optimisation method for SSDs [4], of which the idea is to overlap the CPU computation time with the SSD access time. We plan to integrate both approaches to enhance application launch performance further on SSDs.

Acknowledgments: This research was supported by the WCU (World Class University) program through the National Research Foundation of Korea funded by the Ministry of Education, Science and Technology (R33-10085); RP-Grant 2010 of Ewha Womans University; the Korea Research Foundation Grant funded by the Korean Government (KRF-2008-357-D00208); and the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (20100000479).

6 January 2011
doi: 10.1049/el.2011.0042

J. Ryu and H. Shin (*School of Computer Science and Engineering, Seoul National University, Seoul, Republic of Korea*)

E-mail: shinhs@snu.ac.kr

Y. Joo, S. Park and K.G. Shin (*Department of Computer Science and Engineering, Ewha Womans University, Seoul, Republic of Korea*)

K.G. Shin (*Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI, USA*)

K.G. Shin: is affiliated to both the University of Michigan and the Ewha Womans University

References

1 Park, S.Y., Seo, E., Shin, J.Y., Maeng, S., and Lee, J.: 'Exploiting internal parallelism of flash-based SSDs', *Comput. Archit. Lett.*, 2010, **9**, (1), pp. 9–12

2 Microsoft: 'Support and Q&A for solid-state drives', May 2009, http://blogs.msdn.com/e7/archive/2009/05/05/support-and-q-a-for-solid-state-drives-and.aspx

3 Russinovich, M.E., and Solomon, D.: 'Microsoft Windows internals' (Microsoft Press, 2004, 4th edn), pp. 458–462

4 Joo, Y., Ryu, J., Park, S., and Shin, K.G.: 'FAST: Quick application launch on solid-state drives'. Proc. USENIX Conf. on File and Storage Technologies, San Jose, CA, USA, February 2011, pp. 259–272