DS-Cache: A Refined Directory Entry Lookup Cache with Prefix-Awareness for Mobile Devices

Lei Han[†], Bin Xiao[†], Xuwei Dong[‡], Zhaoyan Shen[*] and Zili Shao[*]
[†]Department of Computing, The Hong Kong Polytechnic University
[‡]School of Computer Science, Northwestern Polytechnical University
[*]School of Computer Science and Technology, Shandong University
[*]Department of Computer Science and Engineering, The Chinese University of Hong Kong
Email: {cslhan, csbxiao}@comp.polyu.edu.hk, dongxuwei@mail.nwpu.edu.cn,
shenzhaoyan@sdu.edu.cn, shao@cse.cuhk.edu.hk

***Abstract*—Our modern devices are filled with files, directories upon directories. Applications generate huge I/O activities in mobile devices. Directory cache is adopted to accelerate file lookup operations in the virtual file system. However, the original directory cache recursively walks all the components of a path for each lookup, leading to inefficient lookup performance and lower cache hit ratio. In this paper, we for the first time fully investigate the characteristics of the directory entry lookup in mobile devices. Based on our findings, we further propose a new directory cache scheme, called Dynamic Skipping Cache, which adopts an ASCII-based hash table to simplify the path lookup complexity by skipping the common prefixes of paths. We also design a novel lookup scheme to optimize the directory cache hit ratio. We have implemented and deployed DS-Cache on a Google Nexus 6P smartphone. Experimental results show that we can significantly reduce the latency of invoking system calls by up to 57.4%, and further reduce the completion time of real-world mobile applications by up to 64%.**

***Index Terms*—Mobile devices, directory cache, lookups**

I. Introduction

Mobile devices, including smartphones and tablets, have become ubiquitous. The amount of traffic that generates from mobile devices had been larger than that from traditional computers. Mobile applications, such as Android applications, access data through file operations. Over the last decade, the storage capacity of smartphones has increased from several gigabytes to several hundreds gigabytes. Reported from [1], average smartphone storage capacity has increased from 13.4GB in 2013 to 60.5GB in 2018. Meanwhile, the number of underlying system files and directories are also growing at a remarkable speed, leading to a more complex hierarchical namespace. Which, in turn, slows down application performance. Therefore, the hardwares including processors and storage medium are not exclusively performance optimization. In this paper, we aim at improving mobile device performance by optimizing the accessing process for file namespace.

Running mobile applications generates huge file I/O activities which are firstly served by the virtual file system (VFS) in which the directory cache (DCache) handles file behavior. Many system calls require DCache to operate, such as *open* and *stat*, and these system calls account for abundant I/O activities. Directory entry (dentry) lookup is a set of operations that walking the namespace tree in DCache, starting the first component of a pathname, then recursively finding the next dentry which is an object with a string name and an inode in memory, finally attaining the last component. However, with the increasing depth of directories and the number of files, the lookup process of namespace tree slows down. Considering the fundamental role of file access in operating systems, the benefits brought from the optimization of file access can achieve fast responsiveness of applications [2].

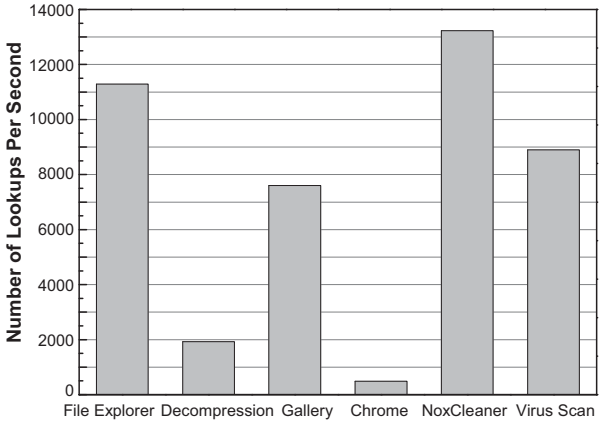


Fig. 1: Path lookup frequency when running the applications.

It is critical to understand what role dentry plays in mobile devices, and how to optimize it to produce a good effect. In the first part, the prior wisdom about the dentry in desktop and laptop devices may not continue to be applicable to mobile devices. Android applications running in a Java virtual machine are mainly managed by the SQLite library, and users have their own habits of using applications and storing files in mobile devices. The Gallery, for example, users readily store pictures with level-by-level directories according to the locations, date, characters, and other classifications. In the second part, several related work have observed that the traditional path lookup scheme needs to be re-examined [3] [4] [5]. Tsai et. al use signatures to accelerate lookup [3], and reduce miss

Corresponding author: Zhaoyan Shen

rates through caching whole directories. They also propose that caching directory completeness can reduce miss rates. Lensing et.al introduce a kernel level file system implementing an efficient metadata placement and direct lookup [4]. Zhan et.al implement full path lookup in a file system to improve access performance [5]. However, most of them need to be integrated with a specific file system, which cannot be moved to mobile devices flexibly and efficiently. Also, the full path hashing approach requires more cache space and hurts the cache hit ratio.

We have performed some preliminary experiments to explore the dentry lookup characteristics of mobile devices. Figure 1 shows the experimental results we have conducted based on a Google mobile phone. It can be observed that the dentry lookups are explosively performed when users run applications. For the junk file cleaning application (NoxCleaner), the number can be 13,121 times per second. We further investigate the lookup latency for a path with different number of components in LMbench benchmark [6]. Table II gives the latency in the system call *stat* on paths with varied depth. We conclude that the latency of dentry lookup is linear in the number of path components, and each component comparison is costly. Furthermore, the majority of dentry lookups share the same prefix at interval when an application is running. For example, when an user opens the Gallery, "/data/com.gallery" is the common prefix of several path lookups. These dentry characteristics in mobile devices motivate us to optimize the original DCache.

In this paper, we propose a novel dentry cache scheme, called Dynamic Skipping Cache (DS-Cache), which is a refined dentry lookup cache with dynamic skipping scheme for mobile devices. Our design rationale is skipping the common prefix of paths to reduce the number of walking component. Specifically, DS-Cache combines the following two techniques. First, DS-Cache maintains an ASCII-based hashtable to cache the dentries of common prefixes. By computing ASCII value of the top several components, there is a great possibility to skip the prefix of a path when performing dentry lookups. Second, DS-Cache employs dynamic skipping scheme to improve DS-Cache hit ratio. DS-Cache dynamically change the number of components of cached prefixes to improve cache hit ratio.

We have implemented and deployed the prototype of DS-Cache on a Google Nexus Android platform. A series of experiments have been conducted to evaluate the performance of DS-Cache. The evaluation results show that DS-Cache can significantly reduce the completion time of real-world mobile applications by up to 64%, and reduce the latency of invoking system calls by up to 57.4%. To summarize, this paper makes the following contributions:

- We observe the characteristics of dentry lookups by presenting a comprehensive experimental study in mobile devices.
- We propose DS-Cache which dynamically utilizes cached common prefixes to simplify the dentry lookups and improve the cache efficiency.

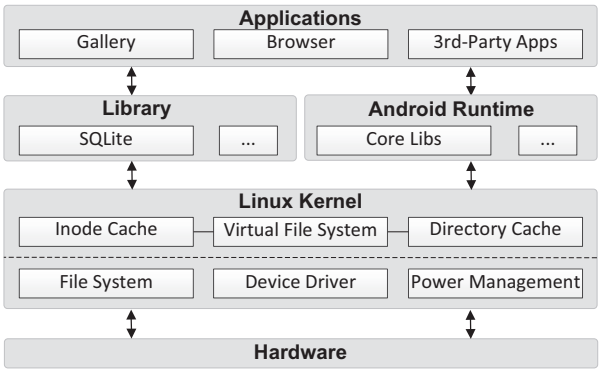


Fig. 2: Android I/O Stack.

- We demonstrate the effectiveness of DS-Cache by conducting a set of experiments.

II. PRELIMINARIES

This section briefly presents background on Android I/O stack, Linux directory cache, and dentry lookup scheme.

A. Android I/O Stack

Android is a Linux-based operating system developed by Google for mobile devices. The storage I/O stack, one of the most complex parts, involves a set of software and hardware layers for data management. Figure 2 shows the I/O stack of Android. Android applications provide various services, including browsing webpages, dialing contacts, etc. Most of applications are managed by the SQLite, which is an in-process library and an embedded SQL database. Android runtime provides core libraries for application runtime environment. Running an application generates quantitative I/O requests, and these requests will be firstly served by VFS at the kernel layer, and then further transferred to individual file system, block layer and driver layer. Finally, the storage devices will response to the I/O requests.

B. Linux Directory Cache

VFS is an abstraction layer on top of individual file system, and it maintains metadata including the DCache which considerably speeds up accesses to commonly used files. A dentry in DCache maintains an object for an in-memory inode. The dentries are organized into a hierarchy which stays the same hierarchy with the on-disk file system. To improve the efficiency of memory space, DCache employs Least Recently Used (LRU) scheme to release invalid or obsolete dentries.

C. Dentry Lookup

The DCache transforms a pathname into a pointer to the associate inode, which involves a lookup operation on a hierarchic structure. Every dentry keeps an unsorted, doubly-linked list of children, and they are organized into a hierarchy. To accelerate lookup, children are kept in a hash table keyed by pathname. Normally, the first lookup starts from the root of a file system (for absolute path). This is well illustrated with the example shown in Figure 3. This absolute path is

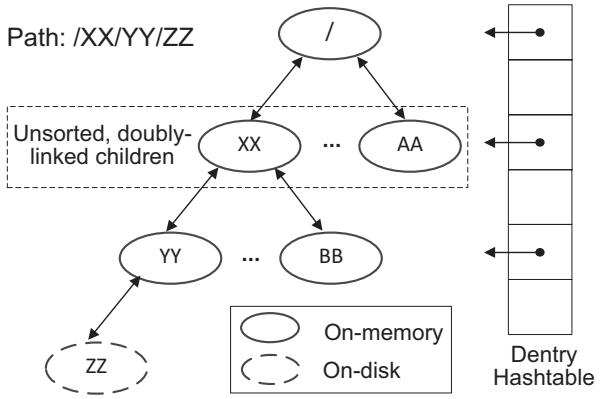


Fig. 3: Path lookup mechanism in DCache.

"/XX/YY/ZZ", and the root has two children, "XX" and "AA". By comparing the first component of the path and the children of root, it yields a pointer for the next component lookup. Suppose the last component "ZZ" is not present in the cache, the DCache needs to ask the file system to finish the lookup; the newly visited inode from disk will be mapped to a new dentry.

III. Dentry Lookup Characteristics in Mobile Devices

In this section, the purpose of our experimental studies is to characterize the dentry lookup in mobile devices. Our experiments are conducted on a Google mobile phone - Nexus 6P. Nexus 6P is equipped with a 2.0GHz oct-core 64 bit Qualcomm Snapdragon 810 processor, 3GB LDPDDR4 RAM, and 32GB NAND flash memory. This mobile phone adopts Android v8.1.0 (Oreo) and Linux v3.10.73.

Table I summarizes the definitions and operations of six popular applications that we select. We collect dentry lookup traces at VFS layer by calculating the number of invoking "link_path_walk()" function related to the corresponding application. The trace is filtered by discarding the dentry lookup of background services when an application is running.

A. Frequent Lookups

Figure 1 shows the average number of path lookups per second. In the collected six traces, the NoxCleaner and File Explorer produce the highest number of lookups, the numbers attend to 13231 and 11285 per second. The Gallery and Virus Scan produce 7600 and 8896 lookups, while the Decompression only produces 1928 lookups. This is because uncompressing a file involves many disk I/O operations which reduce the lookups performance in memory. Note that the execution time of each application is different, since some applications produce file lookups in a blowout type. For example, loading webpages produces 491 lookups per second when users use Chrome, while there are dozens of lookups when users read the information of a preloaded webpage.

TABLE I: Application Details

Applications	Operations
Gallery	Opening the gallery and showing a set of locally stored pictures
Chrome	Browsing multiple websites on the mobile phone
ES File Explorer	Searching any file in the source tree of Linux v3.10.73
Decompression	Extracting the linux-3.10.73.tar.gz to a designated directory
NoxCleaner	Cleaning junk files to reclaim storage space
Virus Scan (360 Security)	Scanning virus on a designated directory

B. Spatial Locality

The dentry lookups present the spatial locality: the majority of neighbouring path lookups share the same common prefix. There are two reasons behind this. First, the related resource files and library files of an application are stored in the same directory or sub-directory. Running an application will issue a number of requests on these files with the common starting paths. Second, the hierarchy-based applications perform a traversal in a breadth-first-search or depth-first-search way. For example, file search approaches a walk of the hierarchy of files, the files in the same sub-directory will be traversed one after another. Therefore, the shared common prefix for the files in the same sub-directory need to be recursively walked during each access.

C. Latency with the Increasing Length

The latency of dentry lookup is linear in the number of path components. We use LMbench to test the latency of system call $stat$ which returns file attributes. LMbench is a suite of simple and portable benchmarks. Table II shows the time spent in $stat$ on varied paths with different numbers of components. We can see that the latencies of paths are linear in the number of components. It is clear to come to this result, one more component lookup incurs more overheads including path name hashing, permission check, and hashtable read. Furthermore, the results show that the latencies of paths with "/data/media/0" without symbolic line and mount point are significantly lower than that of paths with "/sdcard". This is because the symbolic link dentries need to check alias list, and the mount point needs to check mount options.

TABLE II: Latency (s) of $stat$ on paths with different number of components in LMbench benchmark. (The directory "/data/media/0" does not contain symbolic link and mount point, while the "/sdcard" contains both of them. The full paths are "/data/media/0/test/a/b/c/d/e/f" and "/sdcard/test/a/b/c/d/e/f".)

Starting Path	../test	../a	../b	../c	../d	../e	../f
/data/media/0	6.2	6.8	7.6	8.0	8.4	9.2	10.1
/sdcard	32.1	37.2	43.5	48.5	52.5	57.6	63.5

D. Insights for Improvement

The above three characteristics of dentry lookups in mobile devices show us the improvement opportunities. It is easy to see that the improvement can be achieved by reducing the number of components lookups at each path walking, and

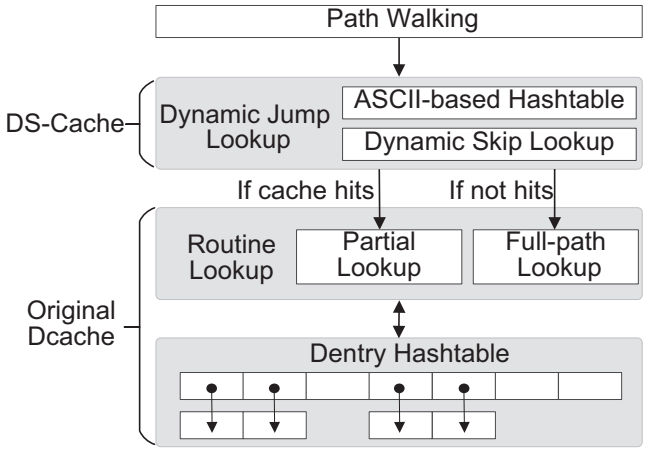


Fig. 4: Architecture of DS-Cache.

by improving directory cache hit ratio. This motivates us to exploit how to build an efficient directory cache scheme for mobile devices.

IV. Dynamic Skipping Cache

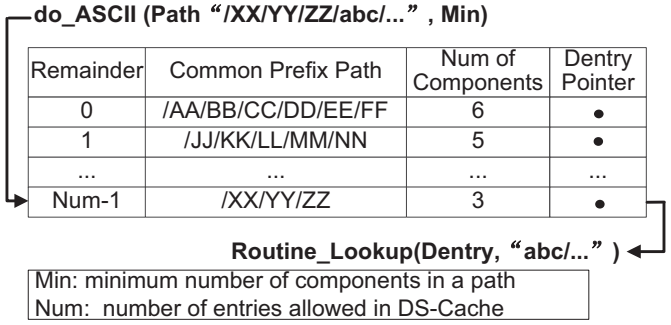
In this section, we present the DS-Cache which dynamically skips a part of components of a path to improve the efficiency of dentry lookups. We first give an overview of DS-Cache, and then present the detailed descriptions of each of its module.

A. Overview

Figure 4 shows the general architecture of the proposed scheme. DS-Cache locates on top of the original DCache. A target path lookup from an I/O activity will be firstly served by DS-Cache, and then passed to the DCache to perform routine lookup according to the caching hit results. An ASCII-based hashtable in DS-Cache caches a few of dentries of the recently-accessed common prefixes. If the entry in the hashtable shares the same prefix with the target path, then the dentry of the prefix for the target will be found. In this way, a target path lookup skips the components of the prefix, and then the rest of the path will be fed to the original DCache for routine lookups. If there is no hit in DS-Cache, the target path is assigned the DCache with the negligible overhead. The basic rationale behind the DS-Cache for accelerating dentry lookup is to skip a number of walking components. To improve the cache hit ratio in DS-Cache, we propose a dynamic skipping scheme that makes a tradeoff between cache hit ratio and the benefit of skipping components. DS-Cache makes dentry lookup performance more consistent in mobile devices.

B. ASCII-based Hashtable

DS-Cache maintains an ASCII-based hashtable to accelerate dentry lookup. There is a great possibility to skip a few components of a path by checking the ASCII-based hashtable. Figure 5 shows the structure of ASCII-based hashtable. A path which has at least "Min" number of components will be performed ASCII-based computation. DS-Cache sums up the ASCII value of each character in the first "Min" number of components, and then takes the remainder when dividing



Remainder	Common Prefix Path	Num of Components	Dentry Pointer
0	/AA/BB/CC/DD/EE/FF	6	•
1	/JJ/KK/LL/MM/NN	5	•
...	...	...	...
Num-1	/XX/YY/ZZ	3	•

Fig. 5: ASCII-based Hashtable in DS-Cache.

by "Num". "Num" means the number of entries allowed in memory. Then an entry that the remainder points to, may share the same common prefix with the target path. According to the number of components in cached prefix, the prefix of the target path will be compared with the cached prefix character by character. If two strings are equivalent, the prefix can be skipped in the parsing of the target path, and then a dentry pointer for cached entry is found for next routine lookup. This is well illustrated with the example shown in Figure 5. Suppose the remainder of example path "/XX/YY/ZZ/abc/..." points to the last entry, and the prefix in the last entry is "/XX/YY/ZZ" which has three components. The DS-Cache then compares the first three components of the example path and the previously stored prefix. If they are equal, the example path lookup skips the first three components, then the dentry pointer of "ZZ" and the remaining example path will be fed to the original DCache for routine lookup.

C. Dynamic Skip Lookup

DS-Cache adopts the dynamic skipping lookup scheme to increase cache hit ratio. Algorithm 1 shows the procedure of dynamic skipping lookup. A target path which has at least "Min" number of components enters DS-Cache, otherwise it performs routine lookup in the original way. The DS-Cache firstly attains the prefix of the target path according to the number of components of currently cached prefix (line 1). If the prefixes of the target path and cached entry are equal, then the target path skips the prefix to walk (line 2-4). Moreover, the entry in DS-Cache will be updated. The number of components of cached entry is added by 1, and the corresponding common prefix path and dentry pointer are updated. This is achieved by DCache which returns the information of next component of the previous stored entry (line 4-5). This maximizes the benefits of skipping components for the future lookups. On the contrary, if there is no hit in DS-Cache, the number of component of cached entry will be deceased by 1, and the cached entry will be replaced by the newly visited prefix. This is because the cached common prefix with a smaller number of path components can increase the hit possibility for next lookups. The lookup scheme is able to dynamically regulate the relation between the DS-Cache hit ratio and the benefits of skipping components.

Algorithm 1 Dynamic Skip Lookup.

Input: Target path tar_path, corresponding entry after ASCII-based computation (current prefix cur_path, num of components num, dentry point p_dentry).
Output: Update the entry in DS-Cache.

1: Attain the prefix of tar_path according to the num
   $pre\_path \longleftarrow$ Get_Path (tar_path, num)
2: **if** pre_path is equal to cur_path **then**
3: Attain remainder of the target path rmd_path of tar_path
   $rmd\_path \longleftarrow$ Get_Last_Path (tar_path, pre_path)
4: Routine_ Lookup (rmd_path, p_dentry)
5: Update the entry
   $num \longleftarrow num$ + 1
   $cur\_path \longleftarrow$ Get_Path (tar_path, num)
   $p\_dentry \longleftarrow$ Get_Dentry (cur_path)
6: **else**
7: Routine_ Lookup (tar_path, NULL)
8: Update the entry
   $num \longleftarrow num$ - 1
   $cur\_path \longleftarrow$ Get_Path (tar_path, num)
   $p\_dentry \longleftarrow$ Get_Dentry (cur_path)
9: **end if**

D. Limitation

Permission changes are not considered in our current design. If the permissions of a cached prefix in DS-Cache is changed by $chmod$ or $rename$, it may incur security issues. To solve the security problems, we simply disable all the entries related to a dentry if the permission of this dentry changes. From the perspective of mobile applications, mobile users rarely change the permission of a directory like computer users.

V. Evaluation

In this section, we present the experimental setup and the evaluation results.

A. Experiment Setup

We have prototyped the proposed DS-Cache on a Google mobile phone - Nexus 6p. The prototype is implemented based on the original DCache, and 468 lines of code are added (9.9% on fs/dcache.c, 8.1% on fs/namei.c, 1.7% on headers, measured by $Sloccount$ [11]). The default "Min" is set to five, and the number of entry "Num" in DS-Cache is set to five in our experiments. These parameters can be configured to accommodate the requirements of users. To highlight the depth and complexity of the directory hierarchy, we put the resources files for applications to the directory "/storage/media/0/database/a/b/c/d/e/f/". This setting matches the current habits of users which categorize files with 6.61 mean directory depth and 15.52 mean maximum depth [12]. We conduct experiments using real-world applications, as shown in Table III. We choose these applications which are file-intensive since the dentry lookups are tightly-coupled with files. The resource files for multimedia applications are stored on the mobile phone in advance. We compare our DS-Cache with the original DCache from various perspectives.

TABLE III: Real-world Application Characteristics

Applications	Resources	Operations
Gallery	9K photos from [7]	Opening app and showing pictures
Music Player	2.8K musics from [8]	Adding musics to a playlist ahead, opening app and automatically loading musics
Reader	18.8K news texts from [9]	Opening app, manually adding a text directory and automatically loading texts
ES File Explorer	48K files from the source tree of Linux v3.10.73	Searching a file on a designated directory
NoxCleaner	System directories	Processing a full scan
AndroBench [10]	SQLite benchmark	Processing a number of queries

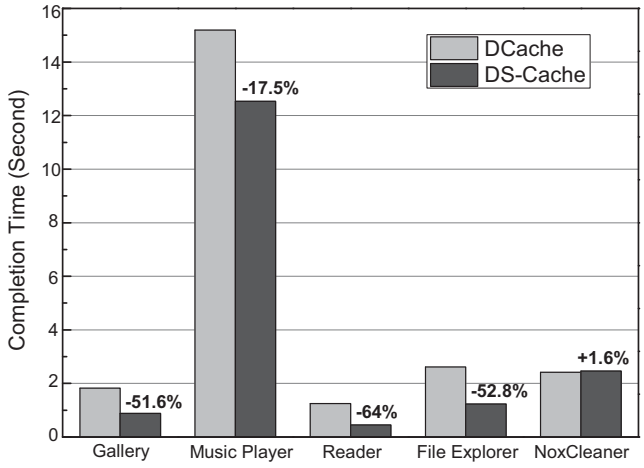


Fig. 6: Comparison of overall performance.

B. Experiment Results

1) Overall Performance: We use the completion time of traversing files and directories when running applications listed on Table III to evaluate the overall performance of the DS-Cache and the original DCache. The results are a mean of five runs. Figure 6 shows the experimental results. It can be observed that DS-Cache significantly reduces the completion time of Gallery, Music Player, Reader and File Explorer. With our proposed DS-Cache, the completion time has a noteworthy reduction, from 17.5% to 64%. This is because our DS-Cache reduces the latency of walking path components, and it improves cache hit ratio by dynamically changing the number of components of the common prefix. Note that, Music Player takes an enormous amount of time to finish traversing resources directories according to the playlist. This is because the back-end operations involve other program operations related to cover art, etc. We note that the performance of NoxCleaner losses 1.6% when compared with the solo DCache. This is because most of system directories have smaller directory depth, while the DS-Cache only serves the paths which have at least five components in this experiment. Furthermore, NoxCleaner skips some unidentified directories to accelerate a full scan, leading no gains with too few components in DS-Cache. Even though, the result shows that DS-Cache only brings negligible overhead under the directories with fewer number of components.

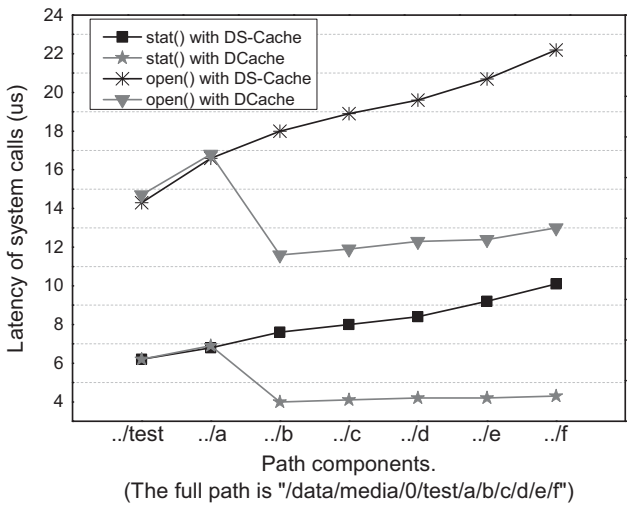


Fig. 7: Comparison of stat/open performance.

We evaluate the SQLite performance by AndroBench, which determines within what time a device processes a number of SQL queries. AndroBench frequently accesses SQLite files in "/data/user/0/com.andromeda.androbench2/databases/". Table IV indicates statistics about queries. DS-Cache performs 4.9% and 1.3% better than the original DCache in insertion and update, respectively. This mainly contains two reasons. First, the frequency of path lookups is low, around 400 times per second, thus the performance improvement from path lookups part is not obvious. Second, the directory depth is not so large that there is only a fraction of gains.

TABLE IV: Transactions per second of an AndroBench test.

	DCache	DS-Cache	Gain
Insertion	307	322	4.9%
Update	292	296	1.3%

2) System Call Performance: We use LMbench to evaluate the system call latency of path lookups with different number of components. Figure 7 shows the latency comparison of invoking $stat$ and $open$ on the sampling paths with DS-Cache and DCache. The sampling paths with the starting "/data/media/0" have no symbolic link and mount point. From the figure we can see that, the lookups performance of DS-Cache and DCache are almost the same when the number of components is no more than five. With the number of components increasing, the latency of both system calls with DS-Cache dramatically reduces, while the latency with DCache is still linear in the number of components. DS-Cache reduces the latency of $stat$ by up to 41.4%, and the latency of $open$ by up to 57.4%. The similar trends can be observed in the directories with symbolic link and mount point.

3) Scalability of DS-Cache: When the scale of entries in DS-Cache increases, more entries that have different common prefixes are involved to further improve the cache hit ratio. We collect average latency when invoking $stat$ system call on ten totally different paths in parallel. Table V evaluates the scalability of DS-Cache. As measured in the table, we observe that the average latency with the larger scale reduces, from 656 microseconds to 252. Moreover, the results indicate that the benefits are not obvious when the number of entries is more than ten, since there are only ten different paths involved in parallel.

TABLE V: Average latency of $stat$ on ten totally different paths in parallel with different number of entries in DS-Cache.

	Num=1	Num=5	Num=10	Num=15	Num=20
Average latency (us)	656	502	266	254	252

VI. Conclusion

In this work, we fully investigate the working characteristics of dentry lookups in mobile devices. Based on our observations, we propose a new directory cache scheme called DS-Cache, which adopts an ASCII-based hashtable to skip the common prefix of paths. DS-Cache consists of two parts: an ASCII-based hashtable that caches the denties of common prefixes; a dynamical scheme that improves cache hit ratio. We have implemented and deployed a prototype on a popular mobile device. The experimental results show that our proposed scheme significantly outperforms the original directory cache.

Acknowledgments

The work described in this paper is partially supported by the grants from the NSFC-Shandong Joint Fund (Analysis and Forecast of Fractal Theory of Common Modeling of Multi-system Pollution Diffusion in Marine Ecological Environment), the Research Grants Council of the Hong Kong Special Administrative Region, China (GRF 15222315, GRF 15273616, GRF 15206617, GRF 15224918), and Direct Grant for Research, The Chinese University of Hong Kong (Project No. 4055096).

References

[1] N. Shah, "Average smartphone nand storage capacity will top 60gb by 2018," Tech. Rep., 2018. [Online]. Available: https://www.counterpointresearch.com/average-smartphone-nand-storage-capacity-will-top-60gb-by-2018/

[2] D. Jeong, Y. Lee, and J.-S. Kim, "Boosting quasi-asynchronous i/o for better responsiveness in mobile devices." in *FAST*, 2015.

[3] C.-C. Tsai, Y. Zhan, J. Reddy, Y. Jiao, T. Zhang, and D. E. Porter, "How to get more value from your file system directory cache," in *Proceedings of the 25th Symposium on Operating Systems Principles*. ACM, 2015.

[4] P. H. Lensing, T. Cortes, and A. Brinkmann, "Direct lookup and hash-based metadata placement for local file systems," in *Proceedings of the 6th International Systems and Storage Conference*. ACM, 2013.

[5] Y. Zhan, A. Conway, Y. Jiao, E. Knorr, M. A. Bender, M. Farach-Colton, W. Jannen, R. Johnson, D. E. Porter, and J. Yuan, "The full path to full-path indexing," in *Proceedings of the 16th USENIX Conference on File and Storage Technologies*. USENIX Association, 2018.

[6] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable tools for performance analysis." in *USENIX annual technical conference*. San Diego, CA, USA, 1996.

[7] L. Fei-Fei, R. Fergus, and P. Perona, "Learning generative visual models from few training examples: An incremental bayesian approach tested on 101 object categories," *Computer vision and Image understanding*, no. 1, 2007.

[8] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur, "Librispeech: an asr corpus based on public domain audio books," in *Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2015.

[9] K. Lang, "Newsweeder: Learning to filter netnews," in *Machine Learning Proceedings 1995*. Elsevier, 1995.

[10] J.-M. Kim and J.-S. Kim, "Androbench: Benchmarking the storage performance of android-based mobile devices," in *Frontiers in Computer Education*. Springer, 2012.

[11] D. Wheeler, "Sloccount," *http://www.dwheeler.com/sloccount/*, 2001.

[12] J. D. Dinneen, "Analysing file management behaviour," *Thesis*, 2017.