A New Metadata Update Method for Fast Recovery of SSD Cache

Jing Yang and Qing Yang
Dept. of Electrical, Computer, and Biomedical Engineering
University of Rhode Island
Kingston RI USA 02881
Email: {jyang,qyang}@ele.uri.edu

***Abstract*—In order to maintain data in an SSD (solid-state disk) cache durable after a crash or reboot, metadata information needs to be stored persistently in SSD. There are two typical metadata methods, update-write-update and write-update. While write-update method has one less SSD write operation than update-write-update for each write I/O, it limits the amount of cached data that can be used after a system crash. We present a design and implementation of a novel metadata update method for SSD cache, referred to as Lazy-Update Following an Update-Write (LUFUW). Our new metadata update method allows maximal amount of data in SSD cache available upon restart after a power failure or system crash with minimal additional writes to SSD. This capability makes restart run twice as fast as existing SSD caches such as Flashcache [1] that can only use dirty data in the cache after crash recovery. We present our prototype implementation on Linux kernel and performance measurements as compared with existing SSD cache solutions.**

Keywords-metadata, SSD, cache, data recovery, storage.

I. INTRODUCTION

Recent developments of flash memory based SSD (solid state disk) provides us with great advantages in terms of high storage performance, low-energy, compact size, and shock resistance. The current price gap and speed difference between hard disk and SSD make SSD a perfect choice as a cache layer between system RAM and disk storage [2-6]. The nonvolatile characteristic of SSD allows the cached data persistent even after power failures or system crashes so that the system can benefit from hot restart. Taking a 500 IOPS disk with 100GB cache system as an example, it will take over 14 hours to fill a cold cache after a system reboot or crash [7]. Therefore, a hot SSD cache after a restart makes a significant difference in storage performance. However, current researches on SSD caches have mainly focused on cache architecture or management algorithms to optimize performance under normal working conditions. Little study is done on exploiting SSD cache's durability across system crashes or power failures.

Hot restart can be achieved by saving parts of cache's mapping and state information (referred to as metadata) in SSD. When system is normally shut down, those metadata can be consistently written back to SSD. However, metadata update is needed to keep cache consistent for recovery purpose after unexpected system crashes or power failures, which is more expensive. There are two ways to update metadata, either using log to record metadata change, or update in-SSD metadata directly. Some SSD cache, such as Flashcache, updates in-SSD metadata synchronously when data is changed to dirty state. Traditionally, an update-write-update operation is needed for each dirty write. The first update indicates which block to be written. The following update confirms the write after the data is successfully written. As a result, one data write requires three SSD write operations. In order to reduce metadata writes to SSD, Flashcache uses write-update instead, i.e. metadata update follows a successful data write. In this case, if there is a crash after a data write but before its metadata update, cache manager does not know which block has been written, giving rise to the possibility that non dirty block may be in an inconsistent state. To guarantee correctness upon recovery, only dirty blocks in the SSD cache can be used. Our recovery experiments have shown that, with this metadata update method, only a small portion of cached data can be used after restart.

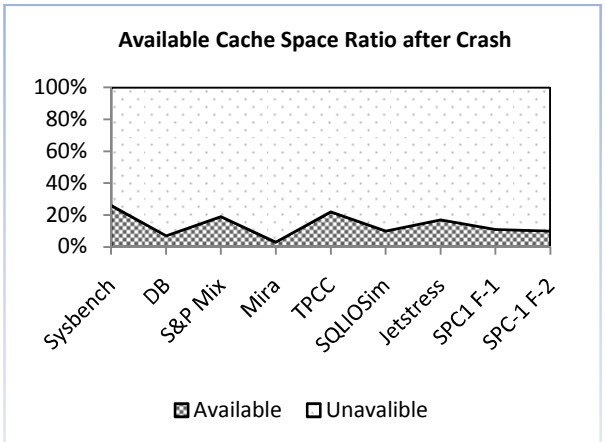

Figure 1. Available cached data upon restart after a crash using write-update metadata method.

To enhance performance with a hot restart, a new metadata update technique is proposed in this paper to maximize useable data upon a system restart in events of crash or power failure. The idea is to write metadata to SSD before writing data for each SSD write

operation. Each metadata entry keeps a flag indicating the corresponding data is in the process of being written to SSD. As soon as the data is successfully written to SSD, this flag is reset in the RAM copy of the metadata. When next time this metadata with the flag reset is written to SSD, it becomes consistent with the data in the SSD cache. Since metadata in SSD is a simple address map, each 4KB page, the basic write unit in SSD, contains over a hundred of metadata entries. Every time there is a metadata update, many such flags are reset in SSD fairly frequently as a free ride. As a result, only a very small portion of cached data in the SSD have this flag set indicating inconsistency and therefore unusable after restart. Since the confirmation updates of metadata are done lazily, we name our metadata update method Lazy-Update Following Update-Write, LUFUW for short. A sliding window and a time stamp are also used to keep track of the unfinished writes. This sliding window and time stamp are written to SSD together whenever a metadata page is written to SSD. In this way, we can further confirm the writes in an older set which does not appear in a newer set's sliding window. A forced flag reset is performed when the window size reaches a predetermined maximal length, further reducing amount of inconsistent data in cache upon restart.

To quantitatively evaluate the performance of our new metadata update method, a prototype has been built on top of Flashcache which appears as a device driver at the generic block layer in the Linux kernel. Standard benchmarks and I/O traces were used to drive the prototype measuring I/O performance in comparison with Flashcache's original metadata management method. Experimental results on standard benchmarks and traces have shown that our new metadata update method allows over 99% of cached data usable at restart as compared to less than 14% with Flashcache's original metadata update method. Because of lazy updates for confirmations that are mostly free ride as piggybacks of other updates, additional metadata overhead is minimal. The new metadata update method makes restart run twice as fast as Flashcache that can only use dirty data in the cache after crash recovery.

In summary, the major contributions of this paper are:

1. A novel metadata update technique has been presented for durable SSD cache. The new technique does not increase metadata update overhead but ensures a hot restart of the SSD cache after a power failure and system crash.
2. A working prototype has been built and tested in Linux Kernel.

The paper is organized as follows. Section 2 gives the design and implementation of the prototype. Experimental settings and workload characteristics are presented in Section 3 followed by results and discussions in Section 4. We discuss related work in Section 5 and conclude the paper in Section 6.

II. PROTOTYPE DESIGN AND IMPLEMENTATION

For a graceful shutdown or restart, metadata in memory can be written into SSD in a consistent state after getting a notification from the system. But for an unexpected system crash or power failure, the metadata may be in an inconsistent state. Therefore, it is necessary to do real time metadata update in SSD for write I/Os. A transaction like update-write-update is the typical and safest way to update metadata. The first metadata update indicates which entry will be written. Then, after the data is successfully written in SSD, the later update confirms metadata complete. If a system crash happens, it is easy to find which entry in the cache is in an inconsistency state. While write-update has been implemented to reduce the two additional metadata update writes to just one, there is a potential danger of data corruption. The reason is, if a crash happens after a data is written but before metadata is updated in SSD, the cache entry contains the data inconsistent with its corresponding metadata (address map and dirty/clean state). That is, the data in that location has been changed but the change has not been reflected in the SSD metadata. To avoid such data inconsistency and guarantee correctness when system restarts, only the blocks that are in dirty state can be used after system restart. No clean blocks can be used because a clean block is not guaranteed to be the same as the one on disk.

In order to maximize consistent data blocks in the cache with minimal overhead of metadata update, our solution is using lazy update. In our method, we update metadata first followed by data write. The confirmation metadata update is done lazily as a free ride of later metadata update in the same metadata block. The overall architecture is shown in Figure 2 with metadata reside in both DRAM and SSD. Figure 3 shows the process of a write in our new LUFUW system. Upon a write I/O, the in-RAM metadata will be updated first to show which entry in the SSD cache will be written. Then the block containing the updated metadata will be writing to SSD as a metadata update write. When the data write finishes, the in memory metadata will be updated again before the write returns to the system. Later, the confirmation update will be reflected to SSD as a free ride when this block of metadata needs to be written to SSD as the first metadata update for other write I/Os.

For each metadata entry, in addition to address map and cache states LUFUW maintains one additional 1-bit flag indicating that the write is in progress. We

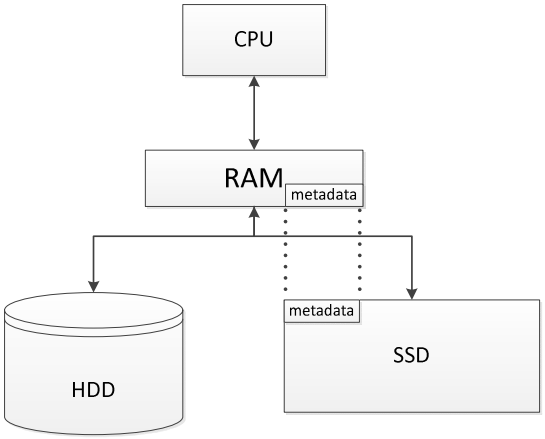

Figure 2. System architecture for LUFUW cache

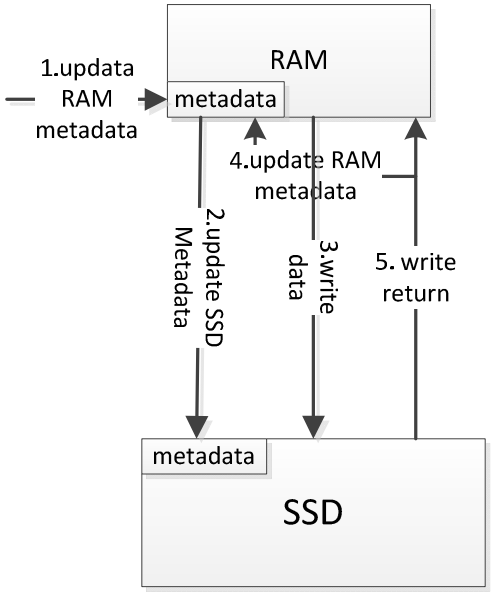

Figure 3. Write process for LUFUW

call it SSD-writing bit. The flag is set to 1 initially when metadata is written to the SSD. As soon as the data is successfully written in SSD, we reset this flag to 0 in the RAM copy of the metadata only. This flag reset will eventually go to SSD through a free ride with other metadata update in the same metadata page, referred to as lazy-update. The metadata state transition is shown in Figure 4. In our design each metadata page contains 240 metadata entries. The chance of a flag being frequently updated in SSD is very high because of data locality. In the worst case, each metadata page will contain at least one unconfirmed. Taking a 320GB SSD cache as an example, there may be over 1 gigabyte space that could not be used after crash. To further increase the valid cache data range after recovery, a sliding window and a time stamp are used to keep track of the unfinished writes. This sliding window and time stamp are written to SSD together whenever a metadata page is written to SSD. The sliding window contains unfinished data writes. When a write comes, the index of the write's metadata block will be inserted into the window. The index will be removed after the write returns. To enhance time consistency, the window is implemented in a similar way as TCP/IP's slide window in network design. The window will not slide without the oldest one being confirmed. In this way, we can further confirm the writes in an older metadata page which does not appear in a newer metadata page's slide window. In our current design, the maximal window length is 63 entries and this window together with a time stamp is piggybacked to every metadata page written to SSD. When system restart after a crash, we only need to find the sliding window with the latest time stamp and make sure their corresponding metadata page's unconfirmed entries and data are not usable. All other data blocks are valid and can serve I/Os. Note that a piggybacked window may be smaller than 63 entries. To prove the concept of our design, we have developed a working prototype based on Flashcache under the Linux kernel.

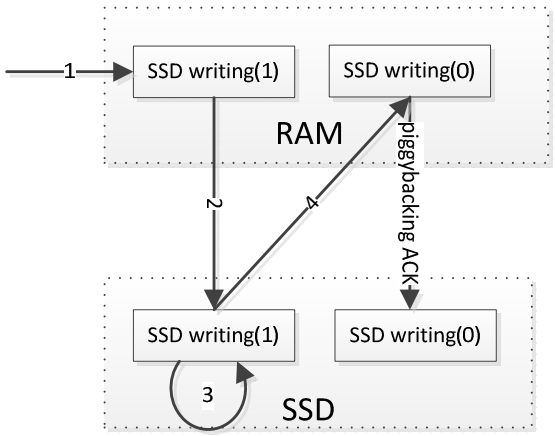

Figure 4. Metadata state transition for LUFUW

We implement our new LUFUW metadata policy in Flashcahe. Flashcache is based on Linux Device Mapper (DM). Unlike LVM2 EVMS and software RAID that combine the address space of all disks, Flashcache map SSD to hard disk and set the address space of SSD to 0. In this way, Flashcache can manage SSD as a cache. It is implemented as a device driver, so it can be easily used in many Linux applications including database system and other applications [8-10]. After Flashcache driver is inserted into Linux kernel, it appears as a device. So Flashcache only processes block I/Os sent to it, and does not affect other system parts. In order to implement our LUFUW metadata update method, we need not only implement our method in Flashcache's write routine, but also in the routine of read misses in Flashcache. Since Flashcache only reuse dirty blocks in the cache after a system crash

or power failure, it does not update metadata in the read miss path because they are not dirty data in SSD cache. In order to make use of maximal amount of cached data upon restart, LUFUW also updates metadata on read misses in SSD.

III. Experimental Setup and Workload Characteristics

A. Experimental Settings

In order to quantitatively evaluate and compare the recovery performance and the overhead of LUFUW with existing cache metadata update policies, we have installed the LUFUW prototype on 3 Dell's PowerEdge R310 rack servers running Linux in our lab. Each rack server has an Intel® Xeon® X3460 processor with 2.80 GHz and 8M Cache. The system RAM of each server is 8GB with 1333MHz and Dual Ranked UDIMM. If system RAM is larger than working data size, most I/O will be cached by system RAM cache. So we restricted RAM used by Linux through GRUB command in different experiments. The hard disk drive used in our tests is 1TB 7.2K RPM SATA drive. We have also installed the original Facebook's Flashcache. We used Flashcache's version 1.0.121. We carried out our experiments on both native and virtualized environment using KVM.

B. Workload Characteristics

The workloads we used in our experiments consist of two standard benchmarks and one real world I/O traces.

The standard OLTP benchmark, SysBench [11], is a modular, cross-platform, and multi-threaded benchmark for systems under intensive I/O loads. We used SysBench in two ways. First we initiated 100 million rows Database (DB) in physical machine server and then tested the DB with 100,000 transactions through 16 threads.

Server consolidation and cloud computing have recently become very popular through virtualizations. In such environment, multiple virtual machines that carry out different applications are installed and run on a single physical server host. To evaluate how such mixed workloads affect the performance of LUFUW cache, we have set up multiple virtual machines (VMs) that ran different benchmarks on one host machine. In such a system, multiple virtual machines running benchmarks generate I/Os to the host hypervisor where our storage drivers are installed. SysBench and PostMark [12] were chosen as typical workloads for their different I/O characteristics. SysBench has high locality while PostMark does not. The storage system receives aggregated I/O requests from all virtual machines. This is an interesting performance test for various storage architectures and is close to real world applications.

A real world trace [13] is also used in the tests. The trace is from a small Microsoft data center. It was collected from 36 different volumes on 13 servers in a period of 7 days. We simulate the data center using six volumes in six virtual machines running concurrently. The six virtual machines all run above the cache so that all virtual machines' I/Os will go through the cache.

IV. Results and Discussions

In this section, we report and discuss our experimental results. To have a fair performance comparison, we run Flashcache with and without the new metadata update mechanism.

Using SysBench as benchmark, we first run 50,000 transactions to the database for warming up the cache. Then we reboot gracefully for normal recovery. We also manually unplugged the power of the machine and restart for crash recovery. After restart, 5,000 transactions are sent to the database to measure the performance. Figure 5 shows the measured results in terms of execution time after system restarts. We used empty SSD cache or cold cache as the baseline for comparison (BL) which is the first bar in the figure. In case of graceful reboot, the entire metadata is updated in the SSD upon receiving a shutdown signal. Therefore, restart performance would be ideal since the cache is warm and the entire cache is useable upon restart. The execution time of graceful reboot is shown as the second bar in Figure 5.

The restart performance of native Flashcache software is shown as the third bar (FC-CR) of Figure 5. Because only the dirty blocks in the SSD cache are useable upon restart after a crash, the restart performance is not much different from cold restart baseline (BL) implying that the cache is pretty cold upon restart. Because of the fact that majority of cached data (clean data) are not useable after restart, we observed only 4% difference between BL and FC-CR. The forth bar in Figure 5 shows the execution time of LUFUW method. We observed substantial performance improvement over traditional metadata update method. The improvement is 92% over cold cache BL. This significant improvement can be attributed to the large amount of useable data in the SSD cache upon restart due to the efficient metadata management technique.

In addition to the standard benchmark, Microsoft small data center traces are used to measure our new metadata update method. We first run 1,200,000 I/Os (200,000 I/Os per virtual machine) to warm up the cache. After this, we reboot or power off the machine in the same way as we did in the database recovery experiment. Then we send another 360,000 I/Os to

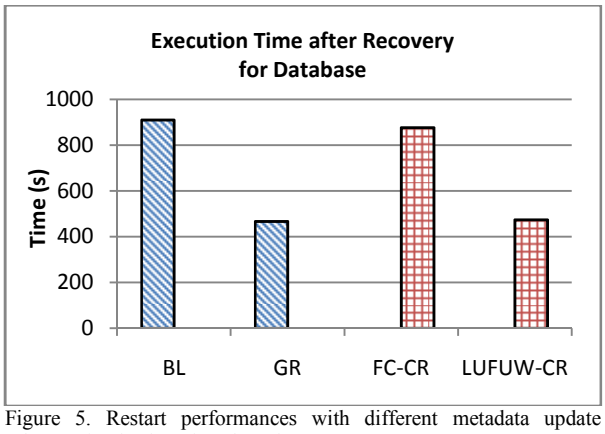

Figure 5. Restart performances with different metadata update schemes for database benchmark.

- BL: baseline, system reboot from an empty cache
- GR: graceful reboot, metadata updated in SSD upon shutdown signal
- FC-CR: Flashcaches' crash recovery
- LUFUW-CR: Crash recovery of LUFUW metadata update mechanism

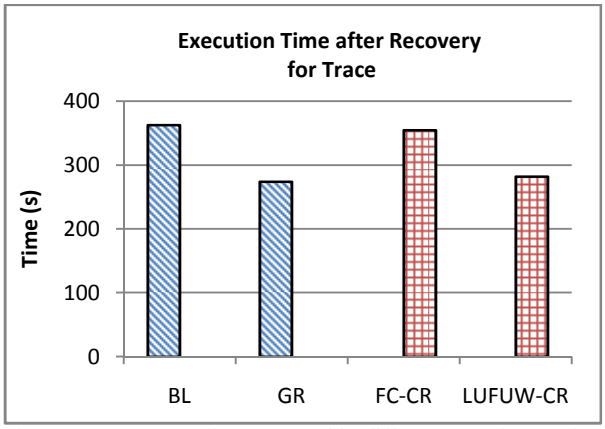

Figure 6. Restart performances with different metadata update schemes for I/O traces.

- BL:baseline, system reboot from an empty cache
- GR: graceful reboot, metadata updated in SSD upon shutdown signal
- FC-CR: Flashcaches' crash recovery
- LUFUW-CR: Crash recovery of LUFUW metadata update mechanism

measure the performance. The result is shown in Figure 6. While the trend in Figure 6 is similar to that of Figure 5, the difference between the new metadata update method and the traditional one is not as dramatic. Our new method is about 20.5% faster than Flashcache's crash recovery and the difference between Flashcache's crash recovery and baseline is about 2%. This is mainly because the trace tests are done in a virtual environment that needs virtual machines (VM) to run. Upon restart after recovery, all VMs where the traces are replayed need to be restarted first. Since the virtual machines are also above the cache, restarting the virtual machines will cause some hot data to be flushed out of the cache reducing the cache hit ratio of restart process. But we still observe significant improvement of our new method over traditional metadata update method.

The next question to be asked is what the cost of such high restart performance would be. We measured the additional writes to SSD as results of metadata updates for both Flashcache and the new LUFUW as shown in Figure 7. As mentioned in Section 2, the number of SSD metadata update writes should increase because LUFUW also updates metadata in the read miss routine. While the result of SysBench shows LUFUW has 12.3% more metadata writes than Flashcache, the mixed workloads of SysBench and PostMark and Microsoft data center traces show just the opposite. In the mixed workloads of SysBench and PostMark, LUFUW has 22.8% less metadata writes than Flashcache. Microsoft data center I/O traces show 48.9% less metadata writes than Flashcache. Let us take a closer look at the results of the mix benchmark to understand why this is happening. While the hit ratios of the two metadata update methods are very close, LUFUW batches 73.8% more metadata update writes than Flashcache resulting in much less total metadata writes to SSD. The reason why LUFUW can batch so much more metadata updates in each metadata page write is two folds. First of all, LUFUW writes metadata to SSD first before data is written, whereas Flashcache write metadata to SSD after data is successfully written to SSD. Therefore, LUFUW collects and batch metadata updates from the I/O queue to the SSD device while Flashcache collects and batch metadata update from returned write I/Os from the SSD device. Because of data locality and high speed CPU, the chance of finding write I/Os that have metadata sharing a same metadata page in the I/O queue is much higher than among returned I/Os that are typically separated by large time interval. The second reason is that we observed a fairly good chance that many overwrites to the same location during the time a metadata page is being updated in the SSD. LUFUW can take full advantage of this write locality while original Flashcache cannot. This is particularly true for Microsoft data center I/O traces due to its high access locality [3]. The time interval between two I/Os in the same LBA is very small for the hot blocks. LUFUW updates metadata first and sets the dirty bit of a write earlier than Flashcache does. This can increase the dirty hit chance of the following write in the same location eliminating unnecessary additional metadata updates and reducing the total the number of metadata updates.

In term of cache performance for normal operations, the two metadata update methods are very close as shown in Figures 8 and 9. In particular, the performance of LUFUW in the mixed benchmark is slightly better than Flashcache. This is because the

proportion of metadata writes is over 15% of the total SSD writes. For the other two benchmarks, metadata writes constitute less than 4% total write I/Os. Our new metadata update method needs more process time resulting in increased the response time. However, the difference is negligible as shown in the figures. The benefits of fast recovery and restart are substantial as demonstrated in our experiments.

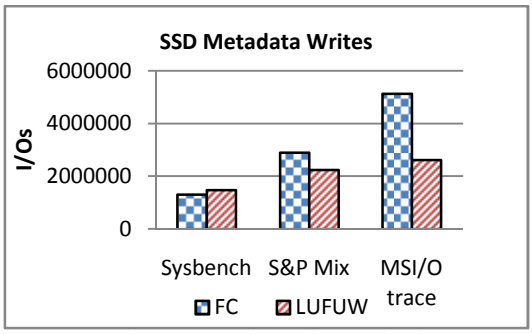

Figure 7. Metadata update overhead

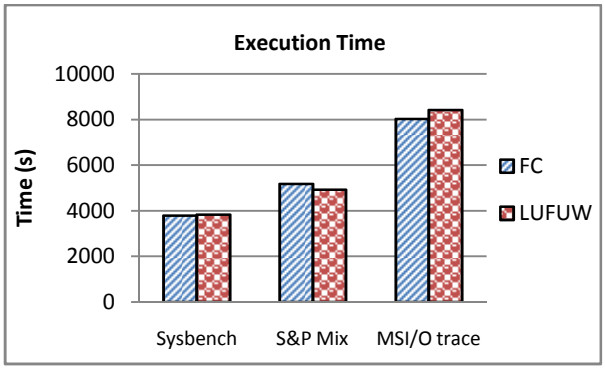

Figure 8. Execution time for three benchmarks

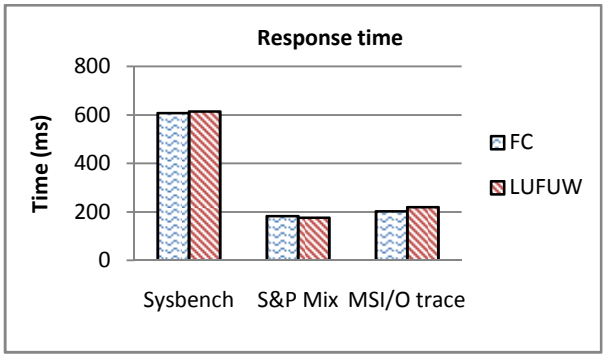

Figure 9. Response time for three benchmarks

V. Related Work

Flash memory SSD has emerged as a promising storage media and fits naturally as cache between system RAM and disk due to its performance/cost characteristics. With fast advances of flash memory technologies, there has been extensive research reported in the literature in improving SSD cache performance [3, 6, 14-17] and reducing write wearing [15, 18-20]. To improve random write performances of SSD, Koltsidas and Viglas have proposed a hybrid SSD and HDD architecture that stores read-intensive pages in SSD and write-intensive pages in HDD [4]. Their online algorithms for optimal page placements have shown great performance advantages over SSD and HDD. Kawaguchi et al [21] developed a UNIX device driver that writes data to the flash memory system sequentially as a Log-structured File System (LFS). The idea has been implemented in today's flash translation layer (FTL) in the controllers of some high end SSDs [22]. Birrell et al [23] presented a technique that significantly improves random write performance of SSD by means of adding sufficient RAM to hold data structures describing a fine grain mapping between disk logical blocks and physical flash addresses. There have been new buffer cache designs that optimize SSD performance by minimizing SSD writes upon cache replacements. Clean First LRU(CFLRU) [24] gives clean pages high priority for evictions until the number of page hits in the working region is preserved in a suitable level. Kim and Ahn proposed Block Padding LRU (BPLRU) [25] buffer cache management algorithm that significantly improves random write performance. A recently proposed design called SieveStore [3] elegantly selects "popular" blocks to store in SSD cache. Two variants of SieveStore, SieveStore-C and SieveStore-D have been provided in their paper. Sieve Store-C uses two levels counters to sieve hot blocks while SieveStore-D keeps the precise access counts of blocks in one level. Canim et al. prototyped a SSD cache as an extension of the in-memory buffer pool named Temperature-Aware Caching (TAC) [9]. Aiming at storing more hot data in SSD, TAC maintains temperature information of disk data. It divides disk into regions, and maintains temperature information in every region as disk accesses occur. Do et al. [26] extends the work of Canim et al. They implemented three SSD design alternatives. But different from TAC, page reads from disk (clean page) are delayed to write to SSD. The write will happen only after the page is evicted from buffer cache. This will reduce writes to SSD and improve performance.

While the above studies aimed at optimizing SSD performance, there is only a few exploiting on SSD cache's nonvolatile feature.

In order to maintain cache data durable after crash or reboot, metadata needs to be stored in persistent storage. In-SSD metadata is persistent and can keep the cache consistent. Typically, there are two ways to update metadata in SSD. One is using log to record metadata changes and the other is updating in-SSD metadata directly upon write I/Os.

The first one is known as logging which is relatively simple. Every log entry may contain cache block's state and the mapping table. Sometimes it also contains a sequence number. Log size increases as write I/Os are issued. In order to restrict the log size, a checkpoint can be placed [7]. Maintaining a log requires cleaning and garbage collection that demand more RAM and CPU resources and may negatively impact system performance. FaCE [10] is an SSD cache using the log like metadata update method. FaCE maintains its metadata similar to the database's log system. Its metadata changes are written to SSD in a single large segment. A checkpoint is used to determine if the segment is in a consistent state.

Direct in-SSD metadata update generally allocates a small part of SSD to store metadata. Metadata is updated directly upon a write. Flashcache [1] is the representative of direct in-SSD metadata update. It updates metadata after data has been written to SSD. Metadata stored in SSD are only updated after a SSD block status changes from clean to dirty or from dirty to clean. Other state changes will not cause data inconsistency. In normal restart case, Flashcache reads the metadata table in SSD and uses all the entries in it to rebuild the cache. In case of crash, only dirty entries in the table are used to rebuild the cache. Flashcache also uses checksums to handle bit corruption and short writes caused by events such as power failures [27] and internal errors. Another work is done by Bhattacharjee et al [8]. They have extended their work in TAC to map the slot directory (metadata) into a flash resident file. As a result, the metadata changes become file changes and are reflected to the file in the cache.

VI. Conclusions

We have presented a new SSD cache metadata update algorithm referred to as LUFUW to make SSD cache durable. Our objective is to maximize cache performance upon a system restart after a crash, power failure, or reboot. LUFUW allows majority of cached data useable when system restarts by means of the new metadata update technique that exploits data locality. LUFUW does metadata update first before data writes. The confirmation update that is essential to guarantee data consistency is done lazily as a free ride with other metadata updates. Such lazy updates substantially reduce SSD write operations whereas keeping the inconsistency window minimal (less than 1% of cached data). A working prototype has been implemented in Flashcache which is a device driver in Linux kernel. Experimental results on standard benchmarks and real world traces have shown that the new metadata update method makes restart run twice as fast as original Flashcache design. Furthermore, metadata processing overhead is kept very low.

References:

[1] Facebook, "Flashcache," https://github.com/facebook/flashcache.

[2] C. Dirik, and B. Jacob, "The performance of pc solid-state disks (SSDs) as a function of bandwidth, concurrency, device architecture, and system organization," *Proceedings of International Symposium on Computer Architecture*, 2009.

[3] T. Pritchett, and M. Thottethodi, "SieveStore: A highly-selective, ensemble-level disk cache for cost-performance," *Proceedings of International Symposium on Computer Architecture*, 2010.

[4] I. Koltsidas, and S.D. Viglas, "Flashing up the storage layer," *Proceedings of the VLDB Endowment*, 2008, pp. 514--525.

[5] H.H. Huang, S. Li, A. Szalay, and A. Terzis, "Performance modeling and analysis of flash-based storage devices," *IEEE Symposium on Mass Storage, Systems Technologies MSST*, 2011, pp. 1 -11.

[6] T. Kgil, D. Roberts, and T. Mudge, "Improving NAND flash based disk caches," *Proceedings of International Symposium on Computer Architecture*, 2008.

[7] M. Saxena, M.M. Swift, and Y. Zhang, "FlashTier: a lightweight, consistent and durable storage cache," *Proceedings of the 7th ACM european conference on Computer Systems*, ACM, 2012, pp. 267-280.

[8] B. Bhattacharjee, K.A. Ross, C. Lang, G.A. Mihaila, and M. Banikazemi, "Enhancing recovery using an SSD buffer pool extension," *Proceedings of the Seventh International Workshop on Data Management on New Hardware*, ACM, 2011, pp. 10-16.

[9] M. Canim, G.A. Mihaila, B. Bhattacharjee, K.A. Ross, and C.A. Lang, "SSD bufferpool extensions for database systems," *Proceedings of the VLDB Endowment*, 2010, pp. 1435-1446.

[10] W.H. Kang, S.W. Lee, and B. Moon, "Flash-based extended cache for higher throughput and faster recovery," *Proceedings of the VLDB Endowment.*, 2012, pp. 1615-1626.

[11] "SysBench," http://sysbench.sourceforge.net/.

[12] J. Katcher, *Postmark: A new file system benchmark*, Technical Report TR3022, Network Appliance, 1997. www. netapp. com/tech_library/3022. html, 1997.

[13] D. Narayanan, A. Donnelly, and A. Rowstron, "Write off-loading: practical power management for enterprise storage," *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, USENIX Association, 2008, pp. 1-15.

[14] Q. Yang, and J. Ren, "I-CASH: Intelligently

coupled array of ssd and hdd," *IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, 2011, pp. 278--289.

[15] L.M. Grupp, A.M. Caulfield, J. Coburn, S. Swanson, E. Yaakobi, P.H. Siegel, and J.K. Wolf, "Characterizing flash memory: Anomalies, observations, and applications," *Proceedings of the Annual International Symposium on Microarchitecture, MICRO*, 2009.

[16] A. Gupta, Y. Kim, and B. Urgaonkar, "DFTL: A flash translation layer employing demand-based selective caching of page-level address mappings," *ACM SIGPLAN Notices*, 2009.

[17] L. Chang, and T. Kuo, "Efficient management for large-scale flash-memory storage systems with resource conservation," *ACM Transactions on Storage*, vol. 1, no. 4, 2005, pp. 381-418.

[18] W.K. Josephson, L.A. Bongo, K. Li, and D. Flynn, "DFS: A file system for virtualized flash storage," *ACM Transactions on Storage*, vol. 6, no. 3, 2010.

[19] G. Soundararajan, V. Prabhakaran, M. Balakrishnan, and T. Wobber, "Extending SSD lifetimes with disk-based write caches," *Proceedings of the 8th USENIX conference on File and storage technologies*, USENIX Association, 2010, pp. 8-8.

[20] A. Gupta, R. Pisolkar, B. Urgaonkar, and A. Sivasubramaniam, "Leveraging value locality in optimizing NAND flash-based SSDs," *Proceedings of of the 9th UISENIX conference on File and stroage technologies(FAST'11)*, USENIX Association, 2011, pp. 7-7.

[21] A. Kawaguchi, S. Nishioka, and H. Motoda, "A flash-memory based file system," *Proceedings of the USENIX 1995 Technical Conference Proceedings*, USENIX Association, 1995, pp. 13-13.

[22] J. Kim, J.M. Kim, S.H. Noh, S.L. Min, and Y. Cho, "A space-efficient flash translation layer for compactflash systems," *IEEE Transactions on Consumer Electronics*, vol. 48, no. 2, 2002, pp. 366-375.

[23] A. Birrell, M. Isard, C. Thacker, and T. Wobber, "A design for high-performance flash disks," *Operating Systems Review (ACM)*, vol. 41, no. 2, 2007, pp. 88-93.

[24] S.Y. Park, D. Jung, J.U. Kang, J.S. Kim, and J. Lee, "CFLRU: A replacement algorithm for flash memory," *CASES 2006: International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, 2006.

[25] H. Kim, and S. Ahn, "BPLRU: a buffer management scheme for improving random writes in flash storage," *Proceedings of the 6th USENIX Conference on File and Storage Technologies*, USENIX Association, 2008, pp. 1-14.

[26] J. Do, D. Zhang, J.M. Patel, D.J. DeWitt, J.F. Naughton, and A. Halverson, "Turbocharging DBMS buffer pool using SSDs," *Proceedings of the 2011 international conference on Management of data*, ACM, 2011, pp. 1113-1124.

[27] M. Zheng, J. Tucek, F. Qin and M. Lillibridge, "Understanding the Robustness of SSDs under Power Fault," *Proceedings of the 11th USENIX Conference on File and Storage Technologies*, USENIX Association, 2013.