http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 12 (1): 71-79, 2013
ISSN 1812-5638 / DOI: 10.3923/itj.2013.71.79

FBSA: A Flash Memory-based Scheduling Algorithm for Optimistic Replication

W. Li, G. Wang, X. Wang and S. Li
Department of Computer Science and Technologies,
Zhejiang University, Hangzhou, China

Abstract: The scheduling algorithm for optimistic replication is important, because it has an extreme effect on replication performance. However, most of the scheduling algorithms are designed for a conventional hard disk device with mechanical disk arms. It may be inefficient when the flash memory which possesses both a higher read/write performance and random access rate is equipped. This study proposed a new flash memory-based scheduling algorithm for optimistic replication: FBSA. It parallelized the write requests on the slave node according to the semantic dependency, thus, full drove the flash memory and improved replication performance. The FBSA was fully implemented under a popular open-source DBMS-MySQL and was proved to show dramatic performance improvement compared with the original scheduling algorithm based on the same hardware and software configurations.

Key words: Scheduling algorithm, flash memory, optimistic replication, semantic dependency

INTRODUCTION

Over the past decades, flash memory has become more and more popular. It was even said that flash technology has the potential to fundamentally change the current storage hierarchy (Debnath *et al*., 2010). Flash memory has so, many amazing features, such as high read/write performance, high random access rates, high shock resistance, high reliability and low power consumption. All of the above take advantage of using the floating-gate transistor to keep information (Pavan *et al*., 1997) and don't suffer from the movement of mechanical disk arms.

However, many applications are not designed for flash memory and could not fully utilize its bandwidth. Simply replacing the hard disk with the flash memory will improve the performance, but flash memory could take us even further, thus, there should be many optimizations available to make the whole system's performance better, such as the parallelism (Chen *et al*., 2011).

Therefore, optimistic replication, also known as lazy replication or asynchronous replication, is a type of technology that could be optimized when it runs on flash memory. Optimistic replication is a widely adopted technology used to improve system availability and performance, especially in geographically distributed data networks. It allows data to be accessed without a priori synchronization, based on the "optimistic" assumption that problems will occur only rarely, if at all. Updates are propagated in the background and occasional conflicts are fixed after they happen. It consists of five stages: operation submission, write set propagation (The write set is the update/write content of a transaction), write set scheduling, conflict resolution and commitment (Saito and Shapiro, 2005). The scheduling portion is very important as it will decide the commit order and affect the final replication performance. In multiple primary systems (in which the update could be taken from multiple nodes), the order is decided by the cooperation of multiple nodes and the conflict resolution is a must needed step. But in prime copy systems (in which the update is taken from the only master node), the order is decided by the commit order on the master node for most applications. Therefore, conflict resolution is not needed anymore and the commitment process is a single applier to apply each write set with a serialized order (Kumar and Satyanarayanan, 1993). This may be efficient enough for a hard disk as the hard disk's bandwidth bottleneck is easy to reach when plenty of write sets are propagated, but will be far away from the flash memory's bandwidth.

In order to full drive the flash memory and thus, improve the overall replication performance, a novel flash memory-based scheduling algorithm-the FBSA is proposed. It would work on the prime copy optimistic replication system and increase the system parallelism.

Corresponding Author: Wei Li, 18F Buynow Building, 23# Jiaogong Rd., Hangzhou, Zhejiang, China
Tel: +86-135-88740963 Fax: +86-571-89731509

FLASH MEMORY AND SCHEDULING ALGORITHM RESEARCH

The exploitation and optimization of flash memory have been extensively studied in recent literature. The PicoDBMS (Pucheral *et al*., 2001) researched the implementation of a DBMS based on a smart card by overcoming its low RAM limitations and the literature (Bolchini *et al*., 2003) discussed how to manage a database on a smart card. However, the smart card was an early stage of the flash memory and most issues on it had been resolved in the enterprise-level flash memory already. The FlashDB (Nath and Kansal, 2007) was designed for sensor networks using flash memory, the features of the traditional DBMS were not considered, such as transaction support. The FAB (Heeseung *et al*., 2006) and the BPLRU (Kim and Ahn, 2008) were the cache management algorithms that helped reduce the random writes to the flash memory. The data page layout was studied (Tsirogiannis *et al*., 2009) which introduced the column-based algorithm that would help attribute aggregation, thus, reducing the data access amounts and increasing the performance. In page logging (Sang-Won and Bongki, 2007) was another important optimization for the flash memory to overcome its heavy in-page over-write cost. Index research was also a very hot topic, BFTL (Wu *et al*., 2007), RBFTL (Xiang *et al*., 2008), LA-Tree (Agrawal *et al*., 2009), ISBF (Lee *et al*., 2007), FD-Tree (Li *et al*., 2010) and UM-B+ Tree (Xu *et al*., 2010) all interested in the index optimization on flash memory and shared the common idea of using the cache system inside the tree to reduce random write operations to the flash memory. All of the above discussions regard flash memory-based optimizations for the DBMS or the DBMS related only, but the flash memory-based optimizations for the database replication are not covered.

On the other hand, the scheduling algorithm, a relatively mature technology, is important for optimistic replication. The first approach is syntactic based, which means the order of the write set is only decided by the time/location. This was not unlike the Active Directory (David, 1999), which was timestamp-based; the LOCUS (Walker *et al*., 1983), which was the vector clock-based; and the Bayou (Petersen *et al*., 1997), which was operation log-based. Likewise, the optimistic replication's scheduling algorithm for MySQL was log based. The syntactic-based scheduling must ensure the total order of submissions and may cause unnecessary conflicts because the semantic dependency was not considered. The second approach is semantic-based, which means the order could be freed by the semantic dependency of each write set. The commutativity was the main property that could be exploited (Jagadish *et al*., 1997), as the commuted write sets could be applied in either order. The IceCube toolkit proposed a concept called constraints, which treated the semantic-based scheduling as an optimization problem (Preguica *et al*., 2003). For the database system, the research was mainly focused on the syntactic method, such as the order insurance for commitment (Daudjee and Salem, 2004) or the snapshot isolation guarantee (Daudjee and Salem, 2006) but did not care too much about semantic-based scheduling. At the same time, none of the replication scheduling algorithms is optimized for the flash memory.

In recent years, some research for the flash memory-based replication has appeared. The Log-structuring cluster (Bernstein *et al*., 2011) was proposed to construct a large pool of flash chips, which acted as a central storage to implement the replication inside. A comparison of the log-structuring flash cluster with the traditional state machine-based replication (Schneider, 1990) also showed its benefit (Malkhi *et al*., 2012). However, a more common flash memory-based replication is seldom discussed.

FLASH MEMORY CHARACTERISTICS

At the same time, the flash memory vendors kept on improving their flash device to overcome many disadvantages by integrating the cache, log-related and other algorithms mentioned above into the FTL (Flash Translation Layer) to make it easier to use. Thus, the flash device could be simply treated as a perfect device with much higher random read/write access rates and be driven better for parallelized access.

The hardware benchmark results would give us a more direct demonstration regarding the characteristics of the flash memory compared to the hard disk. The OCZ RevoDrive 3 PCI-Express SSD (http://www.ocztechnology.com/ocz-revodrive-pci-express-ssd.html) was taken as the flash device and the IOZone (Norcott, 2001) was chosen as the benchmark tool. Figure 1 showed the benchmark result on the flash memory, while Fig. 2 showed the results for the hard disk.

It could be easily seen from the Fig. 1 and 2: For the flash memory, the random write performance would be extremely improved with the parallelization and reached the peak value when the thread number exceeds 4. While the performance of hard disk with parallelization was worse both for the sequential and random write, as the disk arm kept on moving. Then the conclusion is obvious: Unlike the hard disk, a certain degree of parallelization would help improve the performance of the flash memory.

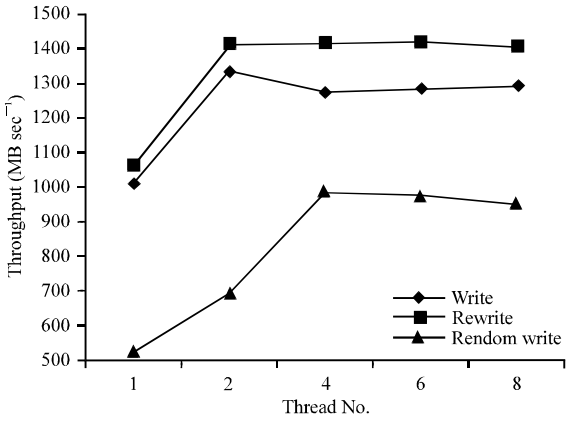

Fig. 1: SSD write performance scaling with thread number (1 GB system memory, 3 GB data write, 16 KB record size)

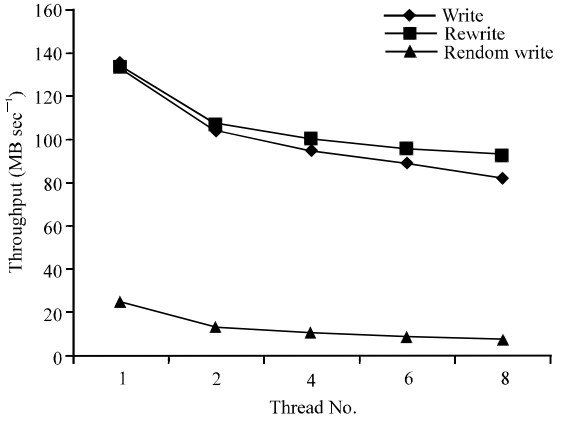

Fig. 2: Disk write performance scaling with thread number (1 GB system memory, 3 GB data write, 16 KB record size)

Table 1: Slave average device utilization

Device	Utilization (%)
Flash memory	25.18
HARD disk	92.26

On the other hand, for the optimistic replication, the MySQL 5.5.20's optimistic replication was taken as an example and the DBT-2 (Wong and Witham, 2011) with 100 data warehouse, 32 connections and zero think time was chosen as the driver workload. After the MySQL replication benchmarks based on the OCZ flash memory or the hard disk were done, Table 1 showed us the average device utilization for the flash device and the hard disk on the slave node during each 30 min' run.

The hard disk was busy, which meant the performance was difficult to be improved, while the flash memory was far away from its bandwidth limitation.

All of the above showed us two conclusions: The parallelization would help full drive the flash memory and there should be some optimization space for the scheduling algorithm of the optimistic replication when it moved from the hard disk to the flash memory.

FLASH MEMORY BASE SCHEDULING ALGORITHM

According to the benchmark of the last section, the parallelization can efficiently improve the flash memory utilization. Then, the new scheduling algorithm is described as follows to improve the performance of optimistic replication with the parallelization.

According to Fig. 3, during the replication process, the master node will propagate the Write Sets (WS) from the master log to the slave node. On the slave side, there are mainly two processes: Dispatching the write sets to multiple commit queues with dependency check and committing the write sets with order adjustment.

The dispatching process will minimize the dependency set's elements number of each write set in all commit queues:

$$\forall w_{ij}: W|0<i<N_j \wedge 0<j<Q.\min(|D_{ij}|) \quad (1)$$

where, w_{ij} is the ith write set in jth commit queue; W is the universal set of all write sets; N_j is the total write set number in the jth commit queue; Q is the total commit queue number; D_{ij} is the dependency set for the write set w_{ij} and ($|D_{ij}|$) stands for making the elements number for D_{ij} minimized.

It is obvious from the above dispatching process that the dependency check is a critical part and the dependency set is a core concept:

$$D_{ij} = \{w_{mn}|0<m<N_n \wedge 0<n<Q.w_{mn} \cap w_{ij} \neq \varnothing\} \quad (2)$$

This definition means each write set w_{mn} inside D_{ij} has overlapped write content with w_{ij} and must be committed ahead. N_n stands for the total write set number in the nth commit queue and the others notions are same with Eq. 1. The construction and maintenance for D_{ij} should be efficient in order to make certain this critical part experiences no performance impact.

At the same time, the commitment process will try best to commit the head element of each commit queue simultaneously. This process will impact the system parallelization too, thus, should be well designed.

The following sections will describe the above two processes in detail.

Dispatching: Dispatching the write set to a proper queue is the key process for scheduling. When w_i stands for the

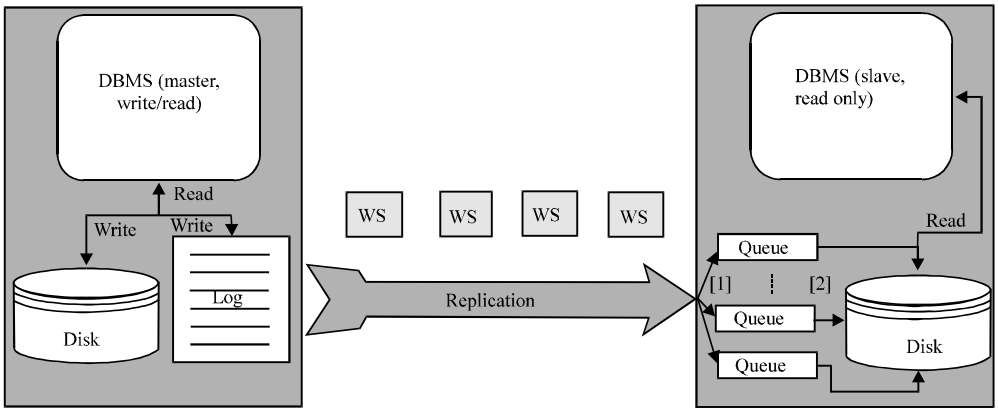

Fig. 3: System architecture for the FBSA enabled replication, 1: Dispatching with dependency check, 2: Commitment with order adjustment

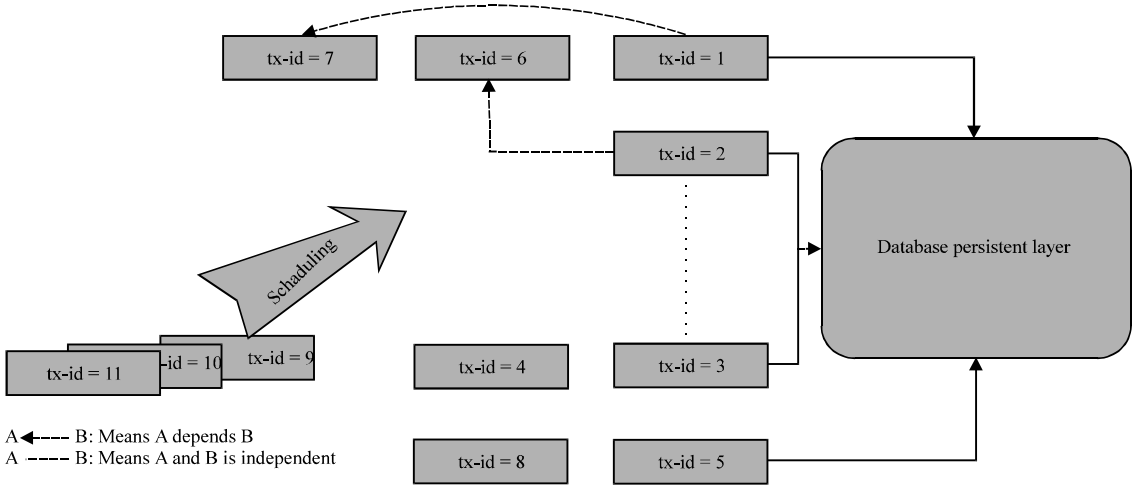

Fig. 4: Dispatching process

new write set that need to be dispatched, Q_n stands for the nth commit queue, Q^1_n stands for the new nth commit queue if w_i is dispatched to, QS stands for the universal queue set, D_i stands for the dependency set of w_i, which is defined in Eq. 2 and the other notions are same with Eq. 1. Three scenarios should be considered:

$$1.\ |D_{ij}| = 0 \tag{3}$$

This means the new write set does not have any dependency with the write sets in the existing queues. This is the best scenario that can be hoped for. The dispatching method should find a Q_n that has the minimal element number and add the new write set w_i in:

$$\exists Q_n : QS \mid 0 < n < Q.\min(|Q_n|) w_i \cup Q_n = Q'_n \tag{4}$$

In Fig. 4, the transaction 8 is this scenario.

$$2.\ |D_i| = 1 \tag{5}$$

This means that there is one and only one write set that is related with w_i. This is also a simple scenario. The dispatching method should find the only queue Q_n that contains the dependency set D_i and add w_i in:

$$\exists Q_n : QS \mid 0 < n < Q.D_i \subseteq Q_n w_i \cup Q_n = Q'_n \tag{6}$$

In Fig 4, the transaction 4 and 7 are this scenario:

$$3.\ |D_i| > 1 \tag{7}$$

This means the new write set has dependency with more than one write in some queues. This is a somewhat complicated scenario. The dispatching method should find a shortest queue is related with w_i and adding w_i in:

$$\exists Q_n: QS|0<n<Q \wedge D_i \cap Q_n \neq \varnothing. \min(|Q_n|)\ w_i \cup Q_n = Q^1_n \quad (8)$$

In Fig. 4, transaction 6 is this scenario.

It can be seen that dependency check is a fundamental part for all of the above scenarios and should be simple and efficient enough to make sure the checking process does not affect the scheduling performance.

Definitions: Let w_i stand for a write set which could be treated as a set of tuples, each tuple should contain a table t_{ij} and the affected record set R_{ij} of this table. T stands for the universal set of all the tables and N_i stands for the table number this write set has:

$$w_i = \{(t_{ij}, R_{ij}) | t_{ij} \in T, 0<j<N_i\} \quad (9)$$

As the table is just the universal set of all records, $R_{ij} \subseteq t_{ij}$ could be got.

In the other words, w_i could also be denoted as a tuple of a table set T_i and a record set R_i which could be used in the dependency check process:

$$w_i = (T_i, R_i | T_i = \{t_{ij}\}, R_i = \{R_{ij}\}) \quad (10)$$

Then, according to the write set's content, a double root data structure is selected inside each queue for dependency check as Fig. 5 shows. Each modified record set will have two parents, one is transaction ID based and the other is table based. From the transaction ID view, each transaction is just denoted as Eq. 10. From the table view, all the tables in this queue is denoted as T_q and each table is denoted as t_m ($t_m \in T_q$), the modified records set for each table t_m is denoted as R_m ($R_m \subseteq t_m$).

Dependency check: When a new write set w_i arrives, its table set T_i will be checked with the table set T_q that the qth queue has, which will be very fast:

- If the intersection set is empty, it is easy to see no dependency with this queue:

$$T_i \cap T_q = \varnothing \quad (11)$$

- If there are some tables in the intersection set:

$$T_i \cap T_q \neq \varnothing \quad (12)$$

check each records set R_m of each table in this intersection set to see if this write set holds any intersections with the records setR_i:

- If no, no dependency either:

$$\forall R_m | R_m \subseteq (T_q \cap T_i) R_m \cap R_i = \varnothing \quad (13)$$

- If yes, there is a dependency here between the current write set and this commit queue:

$$\exists R_m | R_m \subseteq (T_q \cap T_i) R_m \cap R_i = \varnothing \quad (14)$$

As Fig. 5 shows, transaction 3 is a new transaction and it has an intersection table A with transaction 2. If record set M and record set N for table A overlaps, it is the scenario 2.b); otherwise, it is scenario 2.a).

Apply the above steps with each queue. Finally the related write set number is acquired. If this number is zero, apply dispatching method one; if this number is one, apply the dispatching method two and if this number is above one, apply the dispatching method three.

After it is decided to which queue this write set will be dispatched, the write set is appended as a tail node of the commit queue. One thing that should be taken care of is that if the new write set is an independent write set (scenario 1), it should be inserted into the

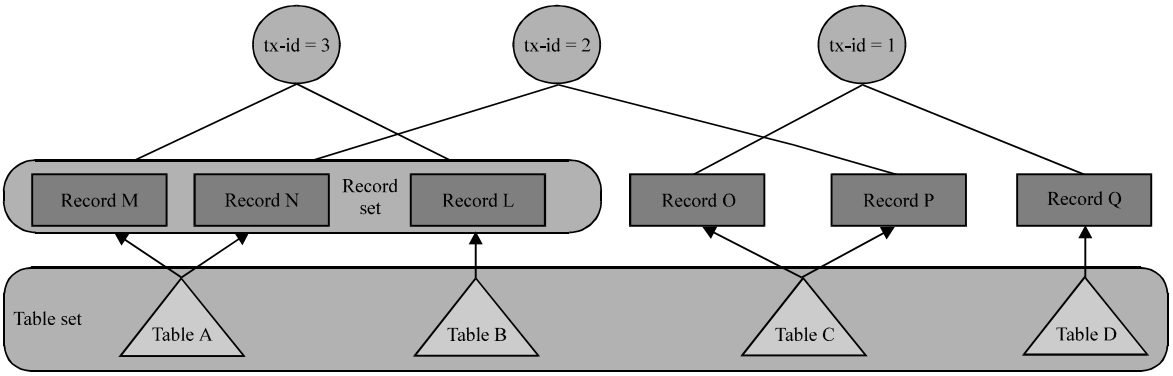

Fig. 5: Dependency check data structure

head position instead of the tail position, as it could be committed without any pre-conditions.

Commitment: While the write set is being dispatched to multiple commit queues, each commit queue is being committed from the head to tail position with order adjustments if needed. Before committing a write set w_i, check if $|D_i| = 0$:

- If yes, apply all the records belonging to this write set and remove this write set from the commit queue. At the same time, remove this write set from each D_m in all queues where $w_i \in D_m$.

$$\forall D_m | w_i \in D_m \rightarrow D_m / w_i = D'_m \quad (15)$$

If no, find an independent write set ($|D_i| = 0$) w_i inside this queue and move it to the head position. If no independent write set could be found, wait for the head write set's dependency set to be emptied and then continue.

With the above processes, the scheduling algorithm could efficiently dispatch and commit each write set during the replication process. This should be an improvement for optimistic replication based on the flash memory, as it will help maximize the parallelization.

PERFORMANCE EXPERIMENTS

The above algorithm was fully implemented under the MySQL 5.5.20 by enhancing the write set scheduling code on the slave node. In this section, a performance comparison was taken between the original MySQL with the traditional scheduling algorithm and the FBSA enhanced MySQL.

The hardware configuration for all the experiments was two 8-core machines with Intel X5670, 8 GB memory, 120 GB HDD (RAID1), 480 GB OCZ RevoDrive 3 PCI-Express SSD (RAID0) and 1 GB Ethernet. The SysBench (Kopytov, 2009) and DBT-2 benchmark tools were used here. The SysBench is a multi-threaded benchmark tool which can quickly get an impression about system performance without setting up complex database benchmarks. The DBT-2 transactional performance test simulates a wholesale parts supplier wherein several workers access a database, update customer information and check on parts inventories. The DBT-2 is a fair usage implementation of the TPC's TPC-C (Raab *et al*., 2001) benchmark specification. Both of them are very popular OLTP benchmark tools for the DBMS.

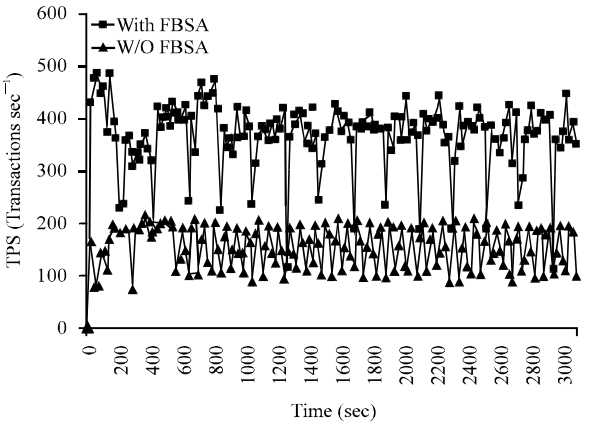

Fig. 6: SysBench slave replication TPS (Transaction Per Second)

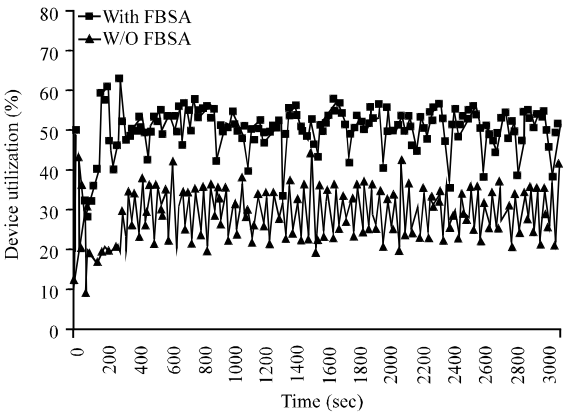

Fig. 7: SysBench slave device utilization

Experiment 1: For the SysBench, the multiple-table was enabled with a 19 GB database, 64 connections, zero think time and 60 min run time. The FBSA was configured with 8 commit queues.

Figure 6 and 7 showed: With the FBSA, the performance was around 2 times better on the slave node and the device was nearly 2 times as busy.

Experiment 2: For the DBT-2, 100 data warehouse was used with a 10 GB database size, 32 connections, zero think time, 12 commit queue number and the other configurations were same with Experiment 1.

Figure 8 and 9 showed: With the FBSA, the performance was more than 2 times better on the slave node and the device was nearly 2 times busy.

Both the above experiments showed that under different workloads, the FBSA could drive the flash memory busier with the parallelization, thus, improved the replication performance by different degree. If the

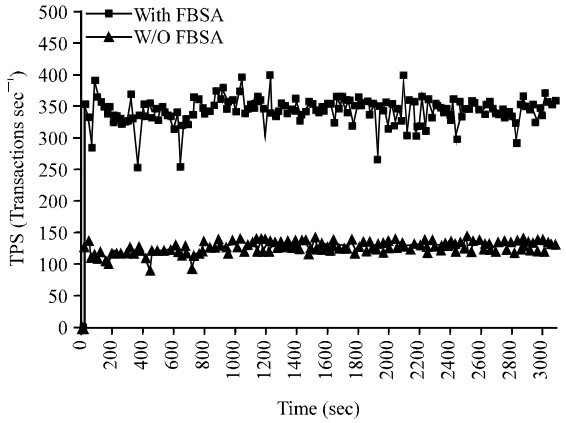

Fig. 8: DBT-2 slave replication TPS

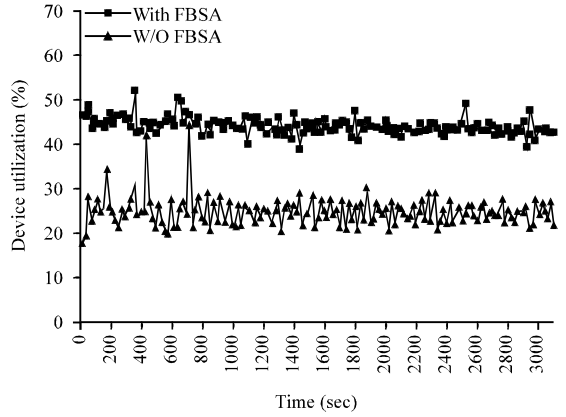

Fig. 9: DBT-2 slave device utilization

workload itself could not be parallelized or only a few operations could be parallelized, the improvement would not be that obvious. But most workloads in the real world could be parallelized just like DBT-2 and multiple-table SysBench. Then, the FBSA should be a practical algorithm for most of workloads.

Experiment 3: The above two experiments used the commit queue number of 8 and 12 separately. This was because each of them could help get the best performance with the benchmark tool being used as shown in Fig. 10 (The sweeping results based on the different commit queue numbers for the SysBench and DBT-2 under the same configurations as Experiment 1 and 2).

Regarding the SysBench, it could be seen from Fig. 10, when the commit queue number was below 8, the performance would scale up as the commit queue number increased. But if the commit queue number kept on increasing, the performance would stop scaling up and even began scaling down. This was caused by the extra

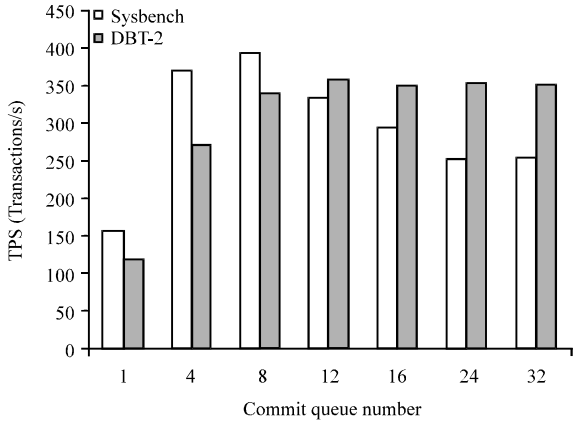

Fig. 10: Slave replication TPS scales with commit queue number

contention brought by the commit queues. The same thing happened for the DBT-2, but the peak number was 12. Therefore, the commit queue number should be chosen carefully in order to obtain the best performance and be adjusted in accordance to different workloads and hardware/software configurations.

CONCLUSION AND FUTURE WORK

From the above discussion and experiments, the conclusion is obvious: The FBSA could help full drive the flash memory and improve the optimistic replication performance by parallelizing the irrelative write requests and committing them simultaneously. The experiments with the SysBench and DBT-2 benchmark tools on the FBSA enhanced MySQL has proved that it is an efficient algorithm and it could improve the overall replication performance by 2-3 times.

The dispatching algorithm in the FBSA is based on a greedy algorithm, which means the decision-making is based only on the current queue's status. A global algorithm that may consider all the write sets in a time window and schedule them with the global dependency relations may help lessen the dependencies between the write sets and could be a future work for this study. Also, the FBSA's best commit queue number is based on the experiment value, predicting upon the best commit queue number based on different hardware/software configurations and workloads may be the future work of this study too.

ACKNOWLEDGMENTS

As advisors, sincerely thanks Prof. Shanping Li for the constructive suggestion and timely encourages. And

also thanks very much for Xingen, Wang's help in the FBSA coding and Gengliang, Wang's effort for the experiments.

REFERENCES

Agrawal, D., D. Ganesan, R. Sitaraman, Y. Diao and S. Singh, 2009. Lazy-adaptive tree: An optimized index structure for flash devices. Proc. VLDB Endowment, 2: 361-372.

Bernstein, P., C. Reid and S. Das, 2011. Hyder-a transactional record manager for shared flash. Proceedings of the 5th Biennial Conference on Innovative Data Systems Research, January 9-12, 2011, Asilomar, California, USA., pp: 9-20.

Bolchini, C., F. Salice, F.A. Schreiber and L. Tanca, 2003. Logical and physical design issues for smart card databases. ACM Trans. Inf. Syst., 21: 254-285.

Chen, F., R. Lee and X. Zhang, 2011. Essential roles of exploiting internal parallelism of flash memory based solid state drives in high-speed data processing. Proceedings of the 17th International Symposium on High Performance Computer Architecture, February 12-16 2011, San Antonio, Texas, USA., pp: 266-277.

Daudjee, K. and K. Salem, 2004. Lazy database replication with ordering guarantees. Proceedings of the 20th International Conference on Data Engineering, March 30-April 2, 2004, Boston, MA., USA.

Daudjee, K. and K. Salem, 2006. Lazy database replication with snapshot isolation. Proceedings of the 32nd International Conference on Very Large Data Bases, September 12-15, 2006, Seoul, Korea, pp: 715-726.

David, I., 1999. Active Directory Services for Microsoft Windows 2000. Microsoft Press, Redmond, WA., USA.

Debnath, B., S. Sengupta and J. Li, 2010. FlashStore: High throughput persistent key-value store. Proc. VLDB Endow., 3: 1414-1425.

Heeseung, J., J. Kang, S. Park, J. Kim and J. Lee, 2006. FAB: Flash-aware buffer management policy for portable media players. IEEE Trans. Consumer Electron,, 52: 485-493.

Jagadish, H.V., I.S. Mumick and M. Rabinovich, 1997. Scalable versioning in distributed databases with commuting updates. Proceedings of the 13th International Conference on Data Engineering, April 7-11, 1997, Birmingham, pp: 520-531.

Kim, H. and S. Ahn, 2008. BPLRU: A buffer management scheme for improving random writes in flash storage. Proceedings of the 6th USENIX Conference on File and Storage Technologies, February 26-29, 2008, San Jose, CA., USA., pp: 1-14.

Kopytov, A., 2009. SysBench: A system performance benchmark. https://launchpad.net/sysbench/

Kumar, P. and M. Satyanarayanan, 1993. Log-based directory resolution in the Coda file system. Proceedings of the 2nd International Conference on Parallel and Distributed Information Systems, January 20-22, 1993, San Diego, CA, USA., pp: 202-213.

Lee, H., S. Park, H. Song and D. Lee, 2007. An efficient buffer management scheme for implementing a B-Tree on NAND flash memory. Proceedings of the 3rd International Conference on Embedded Software and Systems, May 14-16, 2007, Daegu, Korea, pp: 181-192.

Li, Y., B. He, R.J. Yang, Q. Luo and K. Yi, 2010. Tree indexing on solid state drives. Proc. VLDB Endow., 3: 1195-1206.

Malkhi, D., M. Balakrishnan, J.D. Davis, V. Prabhakaran and T. Wobber, 2012. From paxos to CORFU: A flash-speed shared log. Oper. Syst. Rev., 46: 47-51.

Nath, S. and A. Kansal, 2007. FlashDB: Dynamic self-tuning database for NAND flash. Proceedings of the 6th International Conference on Information Processing in Sensor Networks, April 25-27, 2007, Cambridge, UK., pp: 410-419.

Norcott, W., 2001. IOZone filesystem benchmark. http://www.iozone.org/

Pavan, P., R. Bez, P. Olivo and E. Zanoni, 1997. Flash memory cells-an overview. Proc. IEEE, 85: 1248-1271.

Petersen, K., M.J. Spreitzer, D.B. Terry, M.M. Theimer and A.J. Demers, 1997. Flexible update propagation for weakly consistent replication. Oper. Syst. Rev., 31: 288-301.

Preguica, N., M. Shapiro and C. Matheson, 2003. Semantics-based reconciliation for collaborative and mobile environments. Proceedings of the Move to Meaningful Internet Systems, November 3-7, 2003, Italy, pp: 38-55.

Pucheral, P., L. Bouganim, P. Valduriez and C. Bobineau, 2001. PicoDBMS: Scaling down database techniques for the smartcard. VLDB J., 10: 120-132.

Raab, F., W. Kohler and A. Shah, 2001. Overview of the TPC benchmark C: The order-entry benchmark. The Transaction Processing Performance Council. http://www.tpc.org/tpcc/detail.asp

Saito, Y. and M. Shapiro, 2005. Optimistic replication. ACM Comput. Surv., 37: 42-81.

Sang-Won, L. and M. Bongki, 2007. Design of flash-based DBMS: An in-page logging approach. Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data, June 11-14, 2007, Beijing, China, pp: 55-66.

Schneider, F.B., 1990. Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Comput. Survey, 22: 299-319.

Tsirogiannis, D., S. Harizopoulos, M.A. Shah, J.L. Wiener and G. Graefe, 2009. Query processing techniques for solid state drives. Proceedings of the 35th SIGMOD International Conference on Management of Data, June 29-July 2, 2009, Providence, Rhode Island, USA., pp: 59-72.

Walker, B., G. Popek, R. English, C. Kline and G. Thiel, 1983. The LOCUS distributed operating system. ACM SIGOPS Oper. Syst. Rev., 17: 49-70.

Wong, M. and T.D. Witham, 2011. Database test suite. http://sourceforge.net/projects/osdldbt/

Wu, C., T. Kuo and L.P. Chang, 2007. An efficient B-tree layer implementation for flash-memory storage systems. ACM Trans. Embed. Comput. Syst., Vol. 6.

Xiang, X., L. Yue, Z. Liu and P. Wei, 2008. A reliable B-tree implementation over flash memory. Proceedings of the 2008 ACM symposium on Applied computing, March 16-20, 2008, Ceara, Brazil, pp: 1487-1491.

Xu, C., L. Shou, G. Chen, C. Yan and T. Hu, 2010. Update migration: An efficient B+ tree for flash storage. Proceedings of the 15th International Conference on Database Systems for Advanced Applications, April 1-4 2010, Tsukuba, Japan, pp: 276-290.