Evaluating Non-In-Place Update Techniques for Flash-Based Transaction Processing Systems

Yongkun Wang, Kazuo Goda, and Masaru Kitsuregawa

Institute of Industrial Science, The University of Tokyo,
4–6–1 Komaba, Meguro–ku, Tokyo 153–8505 Japan
{yongkun,kgoda,kitsure}@tkl.iis.u-tokyo.ac.jp
http://www.tkl.iis.u-tokyo.ac.jp

Abstract. Recently, flash memory is emerging as the storage device. With price sliding fast, the cost per capacity is approaching to that of SATA disk drives. So far flash memory has been widely deployed in consumer electronics even partly in mobile computing environments. For enterprise systems, the deployment has been studied by many researchers and developers. In terms of the access performance characteristics, flash memory is quite different from disk drives. Without the mechanical components, flash memory has very high random read performance, whereas it has a limited random write performance because of the erase-before-write design. The random write performance of flash memory is comparable with or even worse than that of disk drives. Due to such a performance asymmetry, naive deployment to enterprise systems may not exploit the potential performance of flash memory at full blast. This paper studies the effectiveness of using non-in-place-update (NIPU) techniques through the IO path of flash-based transaction processing systems. Our deliberate experiments using both open-source DBMS and commercial DBMS validated the potential benefits; x3.0 to x6.6 performance improvement was confirmed by incorporating non-in-place-update techniques into file system without any modification of applications or storage devices.

Keywords: NAND Flash Memory, SSD, LFS, Transaction Processing.

1 Introduction

Flash memory is a recently emerging storage device. With price sliding fast, the cost per capacity of flash memory is approaching to that of low-end SATA disk drives. So far flash memory has been widely deployed in consumer electronics even partly in mobile computing environments. Extending the deployment of flash memory to enterprise systems looks a natural attempt. Actually many researchers and developers have been studying the idea of utilizing flash memory for enterprise systems. EMC is trying to incorporate flash-based SSDs into their enterprise-level storage products [3].

One big issue arising for deploying the flash memory to the enterprise systems is that flash memory is quite different from disk drives in terms of the access performance characteristics. Disk drives are mainly comprised of mechanical components, thus random access performance is poor due to the seek and rotational overheads. By contrast, flash memory is a solid-state device, without the mechanical components, yielding high random read performance. However, the flash memory cannot be written in place. When updating the data, the entire erase-block containing the data must be erased before the updated data is written there. Since such erase operations are often very time consuming compared with read/write operations, the random write performance of flash memory is relatively poor. Table 1 summarizes necessary time for each operation in Samsung 4GB flash memory chip [19]. In recent major products, the typical latency of random writes is several milliseconds, being comparable with or sometimes even worse than that of the latest high-end disk drives.

Table 1. Operational flash parameters of Samsung 4GB flash memory chip

Page Read to Register (4KB)	25μs
Page Write from Register (4KB)	200μs
Block Erase (256KB)	1500μs

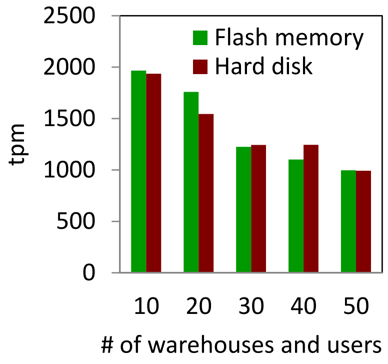


Fig. 1. Performance comparison between disk drive and flash memory (The details are described in Section 4.2)

Due to such a significant performance asymmetry, naive deployment of flash memory into enterprise systems may not exploit the potential performance of flash memory at full blast. Software components of existing enterprise systems are often designed and optimized for disk drives. Fig. 1 shows a typical example: we measured the obtainable throughput by the TPC-C benchmark on a commercial DBMS with the disk drive and flash memory. Contrary to our expectation, we could gain little or sometimes even lose by simply replacing the conventional disk drive with the flash-based SSD in this case study. That is, it may not be easy for existing enterprise system to directly enjoy the potential performance of recent flash memory.

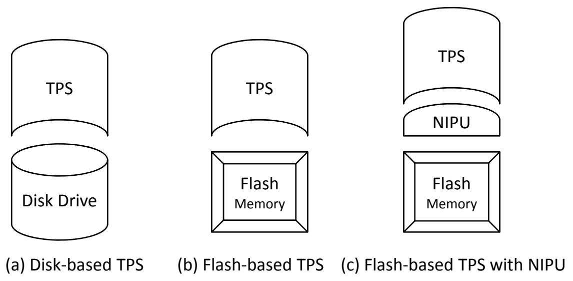


Fig. 2. Comparison of transaction processing system designs

One solution is to redesign the system so that the system can be fully optimized for flash memory. For instance, if we were able to rewrite all the code of database engines and operating systems specially for flash memory, the system could derive the maximum performance. Such a solution may be possible for limited systems. But when it comes to enterprise systems that have a variety of customers, the huge cost of development may not be well accepted by many CIOs. In addition, a variety of succeeding solid-state technologies such as PCRAM [17] are about to emerge. Therefore, it may not be a good choice to invest huge cost on special development only based on flash memory.

Rather, if we could derive reasonable performance improvement of a flash-based enterprise system by simply incorporating optimization techniques into the IO path without modifying other components of the system, as shown in Fig. 2, it could be a good news for many CIOs even though it may not exploit the potential performance of flash memory at 100 percent. This paper studies the effectiveness of non-in-place update (NIPU) techniques through the IO path of transaction processing systems. NIPU techniques can convert a stream of in-place write operations into a stream of non-in-place write operations. Implementation of such techniques between DBMS and flash memory can significantly reduce the number of in-place writes, thus considerably improving the IO throughput of the flash memory. We built an experimental system using both open-source DBMS and commercial DBMS with a conventional hard disk drive and a flash-based SSD and then evaluated the effectiveness of deploying the NIPU techniques into transaction processing systems.

Our measurement-based analysis shows that x3.0 to x6.6 performance improvement can be expected by incorporating NIPU techniques into file systems without any modification of applications or storage devices. To the best of our knowledge, this finding has not yet been reported.

The rest of this paper will be organized as follows: Section 2 will briefly summarize the issue of flash memory for transaction processing system. In Section 3, we will discuss the deployment of the NIPU techniques on flash-based transaction processing system. Our deliberate experiments will be described in Section 4. Section 5 will summarize the related work. Finally, our conclusion and future work will be provided in Section 6.

2 Issue of Flash Memory for Transaction Processing

Unlike the traditional hard disk, which has an approximately symmetric read and write speed, flash memory, on the contrary, has substantial difference between the speeds of read and write, as shown in Table 2. The average response time of read, whatever in sequential or random mode, as well as that of the sequential write, is about two orders of magnitude faster than that of the hard disk. By contrast, the average response time of write in random mode, is comparable or even worse than that of the hard disk. This is primarily because the flash memory cannot be updated in place; a time-consuming block-erase operation has to be performed before the write operation, as disclosed in Table 1 [19]. For the sake of better performance, the size of erase block is usually large, about several hundred KB, leading to an expensive time cost of erase operation compared to that of flash read.

Table 2. Average response time of the flash memory and hard disk with the transfer request size of 4KB. Experiment setup is the same as that in Section 4.1 except here the hard disk and flash memory is bound as the raw device. Benchmark is Iometer 2006.07.27 [6].

	Hard Disk		Flash Memory	
	Read	Write	Read	Write
Sequential	$127\mu s$	$183\mu s$	$94\mu s$	$75\mu s$
Random	$13146\mu s$	$6738\mu s$	$106\mu s$	$8143\mu s$

The poor random write performance of flash memory could be painful for some transaction processing systems. In these systems, the intensive random write is often the main stream of disk IO. Though the operating system has an efficient buffer policy to cache the individual write operations into a bulk update, the performance characteristics of flash memory has been hardly considered here. Therefore, it would be problematic for the existing transaction processing systems to run on the flash memory directly, as reported in [2]. Our experiment also illustrates this points in Section 4.2 that the performance was not improved, even worse than that of the hard disk sometime by directly using the flash memory as the main storage media of data, though the flash memory has fine performance on read and sequential write. A better solution, such as NIPU techniques, is required to fully exploit the benefit of flash memory, as discussed in next section.

3 NIPU Techniques on Flash-Based Transaction Processing System

To utilize the flash memory efficiently, a tactful way is to introduce the NIPU techniques for enterprise system to improve the overall performance. Briefly, the

NIPU techniques convert the logical in-place updates into physical non-in-place updates, using special address table to manage the translation between logical address and physical address. An additional process called garbage collection is required to claw back the obsolete data blocks. A good example of the NIPU technique is the log-structured file system described in [18], with an implementation called *Sprite LFS*. Instead of seeking and updating in-place for each file and Inode, the LFS will collect all write operations and write them into a new address space continuously, as illustrated in Fig. 3. For such a NIPU-based file system, the principal feature is that a large number of data blocks are gathered in a cache before writing to disk in order to maximize the throughput of collocated write operations, thereby minimizing seek time and accelerating the performance of writes to small files. Though the write performance is optimized by some detriment of scan performance [4], this feature is greatly helpful on flash memory to make up for the inefficient random write performance since the random read performance is about two orders of magnitude higher than that of erase operations. The overall write performance is hereby improved.

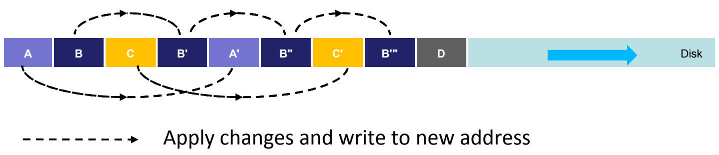


Fig. 3. Non-In-Place Update techniques

Using such techniques for transaction processing systems on flash memory looks a good solution. In this case the flash memory is usually written sequentially through all the way, with a background process reclaiming the obsolete data blocks into the pool of available data block. On the basis of non-in-place update, all the update operations are performed by writing the data pages into the new flash pages, and the erase operations are not required right beforehand as long as the free flash pages are available. Thus, the overall throughput of transactions can be improved.

From a macro view of system, there are several possible places to implement the NIPU techniques through the IO path between DBMS and Flash memory. That is, the NIPU techniques can be incorporated into many places such as Flash Translation Layer (FTL), RAID controller, logical volume manager, file system, and database storage engine. Here arises a problem regarding which place we should implement the NIPU techniques. We are studying on this problem and would like to report it in another paper. In this paper, we focus on the potential benefits of file system. It would be a good choice to load a NIPU-based file system module to OS kernel without any changes to a variety of disk drivers, controllers and database applications.

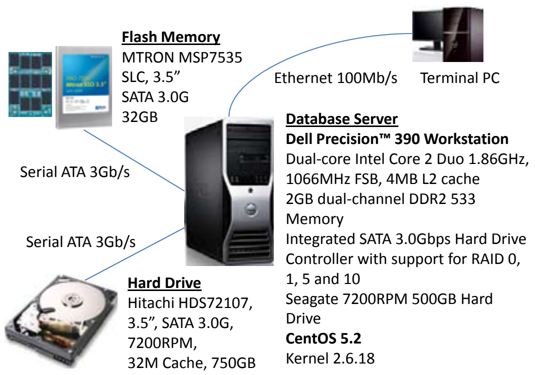
It is to be noted here that a concern on the NIPU techniques is the design and settings of GC (Garbage Collection). Since the NIPU techniques consume free flash pages faster than other methods, the obsolete data pages (garbage) should be reclaimed by fine timing policy to the pool of available data blocks to ensure there are free pages available anytime when there are write requests. We will discuss the influence of the GC settings in Section 4.6.

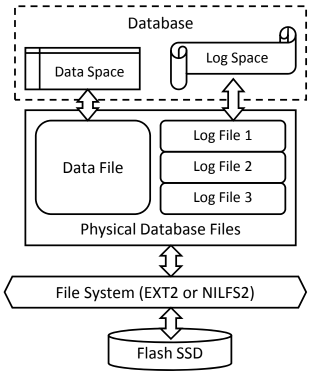
4 Experimental Evaluation

We now describe a set of experiments that validate the effectiveness of the NIPU techniques and compare them against the traditional alternative. We use the popular TPC-C [22] as the benchmark, though it may not exactly emulate the real production workload [5], it discloses the general business process and workload, supported by the main hardware and software database system providers in the industry.

4.1 Experiment Setup

We build a database server on the Linux system. The flash memory is connected to the server with SATA 3.0Gbps hard drive controller as well as the hard disk driver. Fig. 4(a) gives the view of our experimental system.


Flash Memory
MTRON MSP7535
SLC, 3.5"
SATA 3.0G
32GB
Ethernet 100Mb/s
Terminal PC
Serial ATA 3Gb/s
Serial ATA 3Gb/s
Database Server
Dell Precision™ 390 Workstation
Dual-core Intel Core 2 Duo 1.86GHz, 1066MHz FSB, 4MB L2 cache
2GB dual-channel DDR2 533 Memory
Integrated SATA 3.0Gbps Hard Drive Controller with support for RAID 0, 1, 5 and 10
Seagate 7200RPM 500GB Hard Drive
CentOS 5.2
Kernel 2.6.18
Hard Drive
Hitachi HDS72107,
3.5", SATA 3.0G,
7200RPM,
32M Cache, 750GB
(a) System Configuration



(b) System Storage Hierarchy on Flash Memory

Fig. 4. Experiment Setup

We choose a commercial DBMS, as well as popular open source DBMS MySQL [13], as the database system for the TPC-C benchmark. In the commercial database system, the buffer cache is 8MB and log buffer is 5MB, with the block size of 4KB. This block size is set by our previous empirical experiment, in which we performed a low-level disk IO test, with a raw device test program written by us. We find that the optimal IO request size is 4KB for our flash memory. For MySQL, we use *InnoDB* storage engine, buffer cache is 4MB and log buffer is 2MB, with the block size of 16KB. The block size of MySQL is different from that of the commercial DBMS, because MySQL does not allow us to configure the block size, although 16KB might not be optimal.

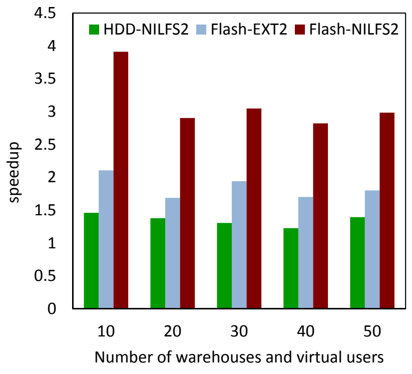
As for the incorporation of the NIPU techniques into the IO path between the databases and devices, we choose a traditional log-structured file system, NILFS2 [16][11], a loadable kernel module without recompilation of the OS kernel, as an intermediate layer between the DBMS and flash memory. As a comparison, we choose the EXT2 file system as the representative of a conventional file system.

The storage hierarchy is simplified and shown in Fig. 4(b). We format the flash memory with EXT2 file system, on which we build the database instance, with all the related files together, such as the data files and log files, as well as the temporary files and system files. Thus, the main IO activities of this instance are confined within the flash memory. We refer this system as "Flash-EXT2". Similarly, we format the flash memory with NILFS2, on which we build the same instance as EXT2 system. We refer this system as "Flash-NILFS2"hereafter. As a comparison, we also build the same system on hard disk, denoted as "HDD-EXT2" and "HDD-NILFS2" respectively.

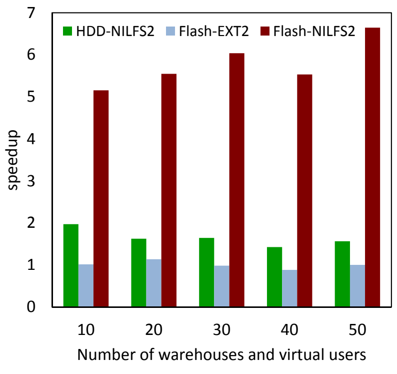
Unlike the EXT2 file system, NILFS2 file sytem has several settings of garbage collection. We set the interval of garbage collection to a very large value to disable this function firstly, so as to simplify the IO pattern. The influence of garbage collection will be discussed in Section 4.6.

4.2 Transaction Throughput

In this test, we create many threads to simulate the virtual users. Each virtual user will have a dedicated warehouse during the execution of transactions. Unlike the real users, virtual users in our test do not have the time for "Key and Think", for the purpose of getting intensive transaction workload. We gradually increase



(a) MySQL



(b) Commercial Database

Fig. 5. Speedup of the transaction throughput on different systems based on "HDD-EXT2"

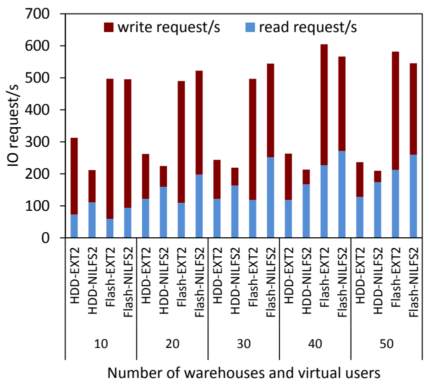
the number of warehouses as well as the number of virtual users to match. The speedup of transaction throughput based on "HDD-EXT2" is shown in Fig. 5.

In Fig. 5(a) we find that speedup of "Flash-EXT2" to "HDD-EXT2" is 1.8–2.1, which means that the naive replacement of flash memory to hard disk could have twofold transaction throughput on MySQL. The speedup of "Flash-NILFS2" shows that the NIPU-based flash memory system can have further improvement, 1.7–1.9 times to "Flash-EXT2", and about 3.0–3.9 times to the "HDD-EXT2". As for the commercial database system shown in Fig. 5(b), it is quite exciting.[1] We can find that the speedup of "Flash-EXT2" to "HDD-EXT2" is around 1.0, showing that the transaction throughput of "Flash-EXT2" is comparable with or sometimes even worse than that of "HDD-EXT2", which verifies our perspective that it is not beneficial for small-size transaction-intensive applications by directly utilizing the flash memory. Remarkably, a significant improvement can be found for "Flash-NILFS2"; the speedup is 5.2–6.6 times to "Flash-EXT2", which manifests that NIPU-based transaction processing system can undergo dramatic improvements on flash memory.

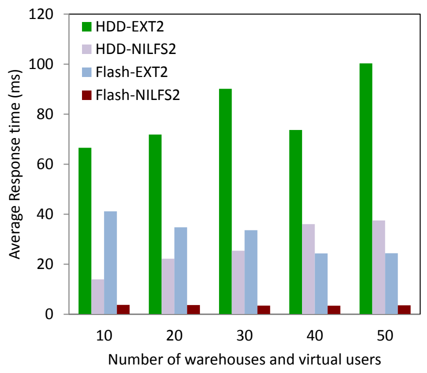
4.3 IO Performance

In our experiments regarding the IOPS, we examine the total number of transfers per second that were issued to the specific physical device. Here a transfer is an IO request to a physical device, and multiple logical requests can be combined into a single IO request to the device. So a transfer is of indeterminate size. Our trace result is shown in Fig. 6. As disclosed in Fig. 6(a) for MySQL, the IO request per second on "Flash-" side is much higher than that of "HDD-" side, which shows that flash memory can improve the IOPS. Meanwhile, the average response time of IO request, as shown in Fig. 6(b), is reduced significantly. Combined with the speedup of transaction throughput in Fig. 5(a), it implies that the NILF2 could coalesce more blocks into a single IO on MySQL, resulting in the higher performance. This can be confirmed in Fig. 6(c), which illustrates IO transfer rate. We can find that the total sector per second on "Flash-NILFS2" is about 6.2–8.4 times as many as that on "HDD-EXT2". On the commercial database system shown in Fig. 6(d), the total IO request per second of "HDD-EXT2" and "Flash-EXT2" is comparable. In sharp contrast, the total IO request per second of "Flash-NILFS2" is outstanding, and the average response time in Fig. 6(e) is also cut down greatly. It implies that the NIPU-based system can handle more requests at a time with shorter service time. Here the average response time includes the time spent by the requests in queue and the time spent servicing them. Since the response time is cut down greatly by NIPU techniques, the OLTP applications, which is required to respond immediately to user requests, could be benefited a lot. The number of sector per second of commercial DBMS shown in Fig. 6(f) follows the same trend as that of the IO request per second, except that on "Flash-NILFS2" it is about 18.6–22.2 times as

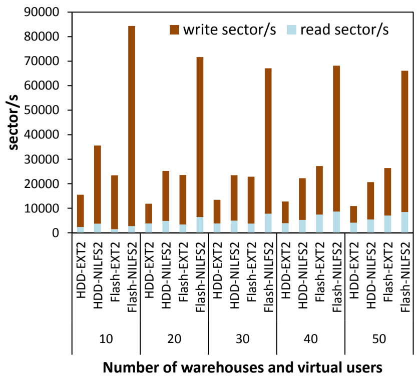
[1] We need further investigation regarding the difference between MySQL and the commercial DBMS.



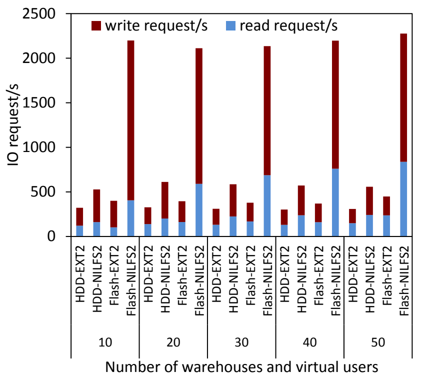
(a) MySQL: Number of IO request per second



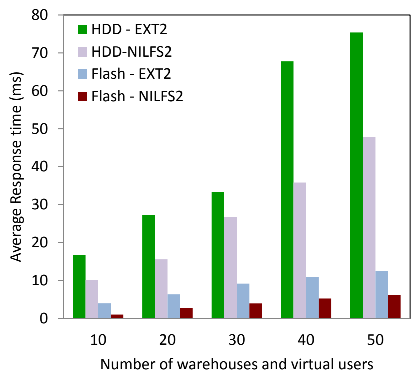
(b) MySQL: Average Response Time of IO Request



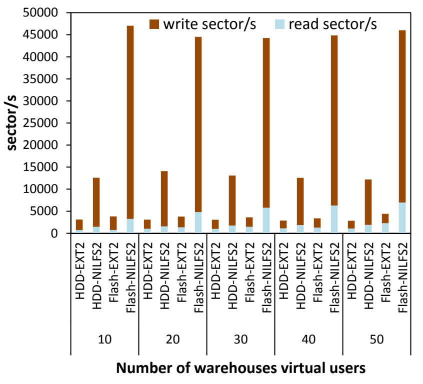
(c) MySQL: Number of sector per second



(d) Commercial DBMS: Number of IO request per second



(e) Commercial DBMS: Average Response Time of IO Request



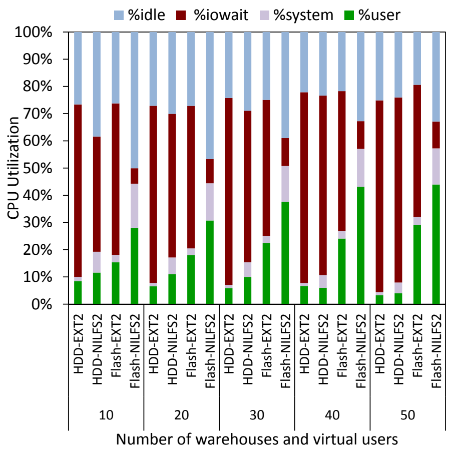
(f) Commercial DBMS: Number of sector per second

Fig. 6. IOPS and Average Response Time

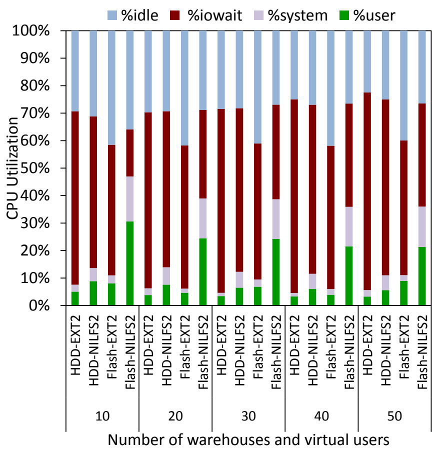
many as that on “HDD-EXT2”. Considered together with Fig. 6(d), the NIPU techniques tends to use relatively large IO request with the increasing of the number of sectors.

4.4 CPU Utilization

In this section we discuss the CPU Utilization in order to analysis the bottleneck of our experimental system. The CPU Utilization is traced when the transactions running in the steady state. The startup and terminate effect is eliminated. Trace result is shown in Fig. 7, in which the CPU Utilization is divided into four portions: *%user, %system, %iowait* and *%idle*. The main portion of CPU time on “HDD-EXT2”, “HDD-NILFS2”, and “Flash-NILFS2”, is spent on waiting for the completion of IO, which implies that the system is possibly “IO-Bound”. However, the CPU Utilization of “Flash-NILFS2” contrasts strongly in the ratio of four portions with the other cases: a uniform distribution of CPU time is observed, caused by the cutback of the CPU time spent on IO wait, and balanced by more CPU time moved to running the user applications, showing that “Flash-NILFS2” can utilize CPU time more efficiently.



(a) MySQL



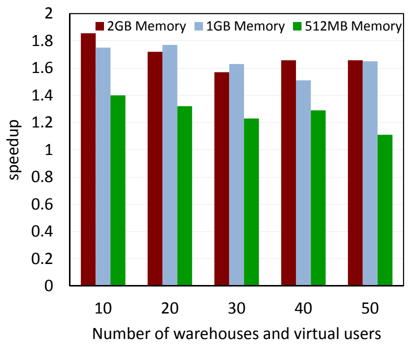
(b) Commercial Database

Fig. 7. CPU Utilization

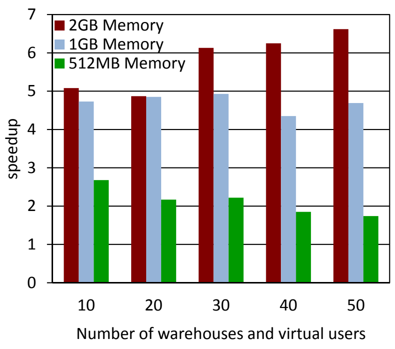
4.5 Disk Buffer Cache

Although we have limited the buffer cache of the database system to a very small size, there is still some influence from the disk buffer cache, as long as we use the file system to manage the data blocks written to the storage device. At this moment, we cannot eliminate the influence of system buffer cache. A passive but efficient approach is to test the system with bound physical memory. Fig. 8

shows the result with 1GB and 512MB physical memory in the same experiment system described in Section 4.1. The *speedup* is the ratio of "Flash-NILFS2" to "Flash-EXT2", i.e. the improvement of NIPU techniques on *flash memory*. For MySQL shown in Fig 8(a), since it is memory efficient, the decreasing is not significant. As for the commercial database shown in the Fig. 8(b), the significant speedup is falling quickly with the very small memory size (512MB). However, with reasonable memory size (1GB), the "Flash-NILFS2" system can gain above fourfold.



(a) MySQL



(b) Commercial Database

Fig. 8. Performance speedup with different amount of physical memory

4.6 Influence of Garbage Collection

We now discuss the influence introduced by the different settings of GC on NILFS2. In Section 4.2 to Section 4.5, no cleaning occurs during the execution, so the measurements represent the best-care performance. In fact, the GC function should be turned on to ensure the free data space. Therefore, the background cleaning processes of GC will consume the CPU time and IO bandwidth, producing some effect to the overall performance of system. A better GC strategy can emulate to the upper level system that the free data blocks are always available, with minimum cost of CPU time.

As indicated in [18], four issues must be addressed regarding the GC: (1) Cleaning Interval (CI), (2) Number of Segments Per Clean (NSPC), (3) Which segments to be clean, and (4) How to group the live blocks. Rosenblum and Ousterhout [18] analyzed and addressed the issue (3) and (4). In this paper, we will focus on analyzing the influence by (1) CI and (2) NSPC.

We can set the protection period (PP) to tell the daemon process how long the segments can be preserved for recovery. The NSPC can also be set when the device is mounted. With these settings, the experiment result in microscopic view is shown in Table 3. we use tuple $(PP, NSPC, CI)$ to denote the detailed settings. With the GC settings shown in Table 3, there is no appreciable change in the tpm (transactions-per-minute) and IOPS (either reads or writes) compared

Table 3. Performance Metrics of NILFS2-based transaction throughput of the commercial database on flash memory with GC

	10 warehouses, 10 virtual users		
	tpm	**IOPS**	**Average Response Time (ms) of I/O Request**
No GC	9983	reads: 406 writes: 1792 total: 2198	1.02
GC(1, 2, 5)	9933	reads: 384 writes: 1856 total: 2240	1.82

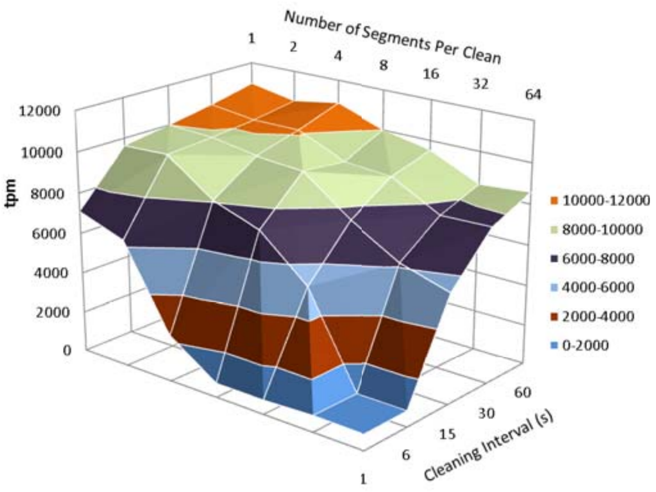


Fig. 9. Transaction throughput with $(0, NSPC, CI)$ on commercial DBMS with 10 warehouses and 10 virtual users

with that without GC. The IOPS includes the IOs issued for GC, so the average response time increases due to the additional IO added by the GC.

We set the protection period to 0, then the obsolete data blocks can be cleaned immediately when the cleaning process is invoked. The transaction throughput with $(0, NSPC, CI)$ is disclosed in Fig. 9. It shows that a greedy cleaning strategy (large NSPC and short CI) will have a detrimental effect to the transaction throughput, although the cleaning is very efficient. The maximum additional IO for GC can be roughly calculated by $\frac{NSPC \times SegmentSize}{CI}$. For example, in Fig. 9, when $NSPC = 4$, the CI should $\geq 30s$ to keep the transaction throughput from falling heavily. We use 8MB segment size,[2] thus the additional IO is about

[2] We use 4KB block size for the NILFS2 file system, and number of blocks per segment is 2048, so the segment size is 8MB.

$\frac{4\times 8MB}{30s} \approx 1.07MB/s$. We should keep the additional IO less than $1.07MB/s$, then the performance will not be degraded. Therefore, carefully choosing the value of $(NSPC, CI)$ and Segment Size with heuristic method would ensure the high transaction throughput as well as the high utilization of the disk cleaned by GC.

5 Related Work

5.1 Non-In-Place Update Techniques

Continuous data protection (CDP) [21][24] is a backup technology automatically saving a copy of every change made to that data to a separate storage location in an enterprise storage system. Another successful example is the Sprite LFS [18], a log-structured file system. The LFS is designed to exploit fast sequential write performance of hard disk, by converting the random writes into sequential writes. However, the side effect is that the sequential reads may also be scattered into random reads. Overall, the performance can be improved to write-intensive applications. The LFS is also expected to improve the random write performance of flash memory, since the fast read performance of flash memory well mitigates the side effect. For the garbage collection of LFS, an adaptive method based on usage patterns is proposed in [15]. Shadow paging [20] is a copy-on-write technique for avoiding in-place updates of pages. It needs to modify indexes and block lists when the shadow pages are submitted. This procedure may recurse many times, becoming quite costly.

5.2 Flash-Based Technologies

By a systematical "Bottom-Up" view, the research on flash memory can be categorized as follow:

Hardware Interface. This is a layer to bridge the operating system and flash memory, usually called FTL (Flash Translation Layer). The main function of FTL is mapping the logical blocks to the physical flash data units, emulating flash memory to be a block device like hard disk. Early FTL using a simple but efficient page-to-page mapping [8] with a log-structured architecture [18]. However, it requires a lot of space to store the mapping table. In order to reduce the space for mapping table, the block mapping scheme is proposed, using the block mapping table with page offset to map the logical pages to flash pages [1]. However, the block-copy may happen frequently. To solve this problem, Kim improved the block mapping scheme to the hybrid scheme by using a log block mapping table [10].

File System. Most of the file system designs for flash memory are based on Log-structured file system [18], as a way to compensate for the write latency associated with erasures. JFFS, and its successor JFFS2 [7], are journaling file systems for flash. JFFS2 performs wear-leveling with the cleaner selecting a block with valid data at every 100th cleaning, and one with most invalid data at other times. YAFFS [23] is a flash file system for embedded devices.

Database System. Previous design for database system on flash memory mainly focused on the embedded systems or sensor networks in a log-structured behavior. FlashDB [14] is a self-tuning database system optimized for sensor networks, with two modes: disk mode for infrequent write, much like regular B^+–tree; log mode for frequent write, employed a log-structured approach. LGeDBMS [9], is a relational database system for mobile phone. For enterprise database design on flash memory, In-Page Logging [12] is proposed. The key idea is to co-locate a data page and its log records in the same physical location.

6 Conclusion and Future Work

For transaction processing system on flash memory, we describe non-in-place update techniques to improve the transaction throughput. In a system based on NIPU techniques, the write operations are performed sequentially; while the GC cleans the obsolete data in the background. This strategy greatly reduces time-consuming erase operations for applications with intensive write operations, thereby resulting in improved overall performance. We use a traditional log-structured file system to build a test model for examination. We then validated NIPU techniques with a set of experiments and showed that the NIPU-based systems can considerably speed up the transaction throughput by x3.0 to x6.6 on flash memory.

In the near future, we plan to apply the non-in-place update technique into different layers of the system and investigate appropriate algorithms for different context.

References

1. Ban, A.: Flash file system. US Patent No. 5404485 (April 1995)
2. Birrell, A., Isard, M., Thacker, C., Wobber, T.: A design for high-performance flash disks. Operating Systems Review 41(2), 88–93 (2007)
3. EMC: White Paper: Leveraging EMC CLARiiON CX4 with Enterprise Flash Drives for Oracle Database Deployments Applied Technology (December 2008)
4. Graefe, G.: Write-Optimized B-Trees. In: VLDB, pp. 672–683 (2004)
5. Hsu, W.W., Smith, A.J., Young, H.C.: Characteristics of production database workloads and the TPC benchmarks. IBM Systems Journal 40(3), 781–802 (2001)
6. Iometer, `http://www.iometer.org`
7. JFFS2: The Journalling Flash File System, Red Hat Corporation (2001), `http://sources.redhat.com/jffs2/jffs2.pdf`
8. Kawaguchi, A., Nishioka, S., Motoda, H.: A Flash-Memory Based File System. In: USENIX Winter, pp. 155–164 (1995)
9. Kim, G.J., Baek, S.C., Lee, H.S., Lee, H.D., Joe, M.J.: LGeDBMS: A Small DBMS for Embedded System with Flash Memory. In: VLDB, pp. 1255–1258 (2006)
10. Kim, J., Kim, J.M., Noh, S.H., Min, S.L., Cho, Y.: A space-efficient flash translation layer for CompactFlash systems. IEEE_J_CE 48(2), 366–375 (2002)

11. Konishi, R., Amagai, Y., Sato, K., Hifumi, H., Kihara, S., Moriai, S.: The Linux implementation of a log-structured file system. Operating Systems Review 40(3), 102–107 (2006)
12. Lee, S.W., Moon, B.: Design of flash-based DBMS: an in-page logging approach. In: SIGMOD Conference, pp. 55–66 (2007)
13. MySQL, `http://www.mysql.com/`
14. Nath, S., Kansal, A.: FlashDB: dynamic self-tuning database for NAND flash. In: IPSN, pp. 410–419 (2007)
15. Neefe, J.M., Roselli, D.S., Costello, A.M., Wang, R.Y., Anderson, T.E.: Improving the Performance of Log-Structured File Systems with AdaptiveMethods. In: SOSP, pp. 238–251 (1997)
16. NTT: New Implementation of a Log-structured File System, `http://www.nilfs.org/en/about_nilfs.html`
17. Pirovano, A., Redaelli, A., Pellizzer, F., Ottogalli, F., Tosi, M., Ielmini, D., Lacaita, A.L., Bez, R.: Reliability study of phase-change nonvolatile memories. IEEE_J_DMR 4(3), 422–427 (2004)
18. Rosenblum, M., Ousterhout, J.K.: The Design and Implementation of a Log-Structured File System. ACM Trans. Comput. Syst. 10(1), 26–52 (1992)
19. Samsung: K9XXG08XXM Flash Memory Specification (2007)
20. Shenai, K.: In: Introduction to database and knowledge-base systems, p. 223. World Scientific, Singapore (1992)
21. Strunk, J.D., Goodson, G.R., Scheinholtz, M.L., Soules, C.A.N., Ganger, G.R.: Self-Securing Storage: Protecting Data in Compromised Systems. In: OSDI, pp. 165–180 (2000)
22. TPC: Transaction Processing Performance Council: TPC BENCHMARK C, Standard Specification,Revision 5.10 (April 2008)
23. YAFFS: Yet Another Flash File System, `http://www.yaffs.net`
24. Zhu, N., Chiueh, T.: Portable and Efficient Continuous Data Protection for Network File Servers. In: DSN, pp. 687–697 (2007)