GREEN COMPUTING
A Case for Data Caching and Flash Disks?

Karsten Schmidt, Theo Härder, Joachim Klein and Steffen Reithermann
University of Kaiserslautern, Gottlieb-Daimler-Str., 67663 Kaiserslautern, Germany

Keywords: Flash memory, flash disks, solid state disk, data caching, cache management, energy efficiency.

Abstract: Green computing or energy saving when processing information is primarily considered a task of processor development. However, this position paper advocates that a holistic approach is necessary to reduce power consumption to a minimum. We discuss the potential of integrating NAND flash memory into DB-based architectures and its support by adjusted DBMS algorithms governing IO processing. The goal is to drastically improve energy efficiency while comparable performance as is disk-based systems is maintained.

1 INTRODUCTION

Recently, green computing gained a lot of attention and visibility also triggered by public discussion concerning global warming due to increased CO2 emissions. It was primarily addressed by enhanced research and development efforts to reduce power usage, heat transmission, and, in turn, cooling needs of hardware devices, in particular, processor chips using extensive hardware controls. Thermal management, however, is a holistic challenge. It includes not only simultaneous optimizations in the materials, devices, circuits, cores, and chip areas, but also combined efforts regarding system architecture (e.g., integration of more energy-efficient storage devices), system software (e.g., energy-optimal application of algorithms), and system management (e.g., a control center responding to workload changes by allocating/deactivating entire servers). In the area of database management systems (DBMSs), up-to-date little research work has contributed to this important overall goal. But, NAND flash memory (also denoted as solid state disk) seems to have the potential to become the future store for permanent database data, because – compared to magnetic disks (disk, for short) – it promises breakthroughs in bandwidth (IOps), energy saving, reliability, and volumetric capacity (Gray and Fitzgerald, 2007).

So far, flash memory was considered ideal for storing permanent data in embedded devices, because it is energy efficient, small, light-weight, noiseless, and shock resistant. So, it is used in personal digital assistants (PDAs), pocket PCs, or digital cameras and provides the great advantage of zero-energy needs, when idle or turned off. In these cases, flash use could be optimally configured to typical single-user workloads known in advance. However, not all aspects count for DB servers and not all DBMS algorithms can be directly applied when processing data on flash.

2 NAND FLASH-BASED DISKS

Because NAND flash memory is non-volatile, allows for sequential and random block reads/writes, and keeps its state even without energy supply, it can be compared to disks and can take over the role of disks in server environments. Therefore, we call such storage units NAND flash-based disks or *flash*, for short. To evaluate their potential when mapping DB data to such devices, we briefly sketch – for DBMS use – the typical *read/write model* of disk and flash.

2.1 Read/Write Models

Disks are devices enabling very fast sequential block reads and, at the same time, equally fast writes, whereas random block read/writes are much slower (requiring substantial “mechanical time fractions”). The block size can be configured to the needs of the DBMS application with page sizes typically ranging between 4KB and 64KB. To hide the access gap between memory and disk, DBMS use a large DB cache in memory (RAM) where (in the simplest

Schmidt K., Härder T., Klein J. and Reithermann S. (2008).
GREEN COMPUTING - A Case for Data Caching and Flash Disks?.
In *Proceedings of the Tenth International Conference on Enterprise Information Systems - DISI*, pages 535-540
DOI: 10.5220/0001725805350540

case) each cache frame can keep a DB page which, in turn, can be mapped to a disk block. In most DBMSs, propagation of DB pages follows the update-in-place principle applying WAL (Gray and Reuter, 1993).

Flash storage is divided into *m* equal blocks typically much larger than DB pages. A flash block normally contains *b* (32 – 128) fixed-size pages where a page ranges between 512B and 2KB. Because zeros cannot be directly written to a page, one must erase (reset) the block to all 1's, before a page can be written. Thus, a written page cannot be updated anymore, but only freshly written after the entire block is erased again. Hence, the block is the *unit of erasure* automatically done by the flash device when allocating an empty block. The page is the smallest and the block the largest *unit of read* whereas the page is the *unit of write*; using chained IO, the DBMS, however, can write $1 < i \leq b$ pages into a block at a time. Note, whenever a page is written *in-place*, the flash device automatically allocates a new block and moves to it all pages from the old block together with the updated page (keeping a cluster property). This *wear leveling* (Ban, 2004) is entirely transparent to the client, i.e., the DBMS, such that all references to displaced pages, e.g., index pointers and other links, remain valid.

Another concern called write endurance and often cited in the literature is the limited number of erase cycles, between 100,000 (older references) and 5,000,000 (most recent references). When a block reaches this erase cycle limit, it cannot be longer used and has to be marked as corrupted. Hence, management of flash relies on a pool of spare blocks; due to the application of wear leveling overly frequent overwriting of the same block is avoided.

2.2 Flash Potential

To gain a deeper and more complete comparative picture of disk and flash, we want to outline the differences and advantages for IO, power, size, and price of both device types and indicate where drastic processing improvements and costs can be anticipated. Here, we can only summarize the evaluation of others (Gray and Fitzgerald, 2007, Nath and Kansal, 2007) in a coarse way and give indicative numbers or orders of magnitude of gains or degradations. Of course, this discussion assumes that DBMS algorithms provide adjusted mappings to take full advantage of the flash potential. For the performance figures, we assume what technology currently provides for fast disks (e.g., SCSI 15k rpm) and flash (SAMSUNG 2008).

IO performance: We distinguish different forms of IO processing: *Sequential IO* continuously reads/writes blocks to the device whereas *random IO* can be directly performed to/from any given block address.

Sequential reads and writes on flash having a bandwidth of ~90 MBps are comparable to those on fast disks.

- Sequential reads and writes on flash having a bandwidth of ~90 MBps are comparable to those on fast disks.
- *Random reads* on flash are spectacularly faster by a factor of 10–15 (2800 IOps compared to <200 IOps).
- *Random writes*, requiring block erasure first, perform worst with ~27 IOps and are slower by a factor of 4–8 compared to disks.

Hence, dramatic bandwidth gains are obtained for random reads while random writes are problematic and have to be algorithmically addressed at the DBMS side.

Energy consumption: The power needed to drive a flash read/write is 0.9 Watt and, hence, by a factor of >15 lower than for a disk. Using the figures for IOps, we can compute IOps/Watt as another indicator for energy-saving potential. Hence, 3,100 flash-reads and 30 flash-writes can be achieved per Watt, whereas a disk only reaches 13 operations per Watt.

Unit size: Starting in 1996, NAND flash chips doubled their densities each year and currently provide 64 Gbit. According to Hwang (2006), this growth will continue or accelerate such that 256 GByte per chip are available in 2012. Because several of these chips can be packaged as a "disk", the DB community should be prepared for flash drives with terabyte capacity in this near future. Hwang (2006) also expects the advent of 20 TByte flash devices in 2015. Of course, disks will also reach a comparable capacity, but flash drives will provide further important properties, as outlined in Section 1.

Price per unit: Today, flash is quite expensive. A GByte of flash memory amounts to 20$, but technology forecast predicts a dramatic decrease to only 2$/GByte in the near future. Therefore, Gray and Fitzgerald (2007) expect that disk and flash of comparable capacity will have roughly the same price (e.g., ~500$ for an SCSI and ~400$ for a flash drive). This assumption allows us to compute IOps/$ as an additional measure of comparison. While flash-read gets 7.0 IOps/$, flash-write gets poor 0.07 and an SCSI operation 0.5 IOps/$. Hence, using

flash, we achieve for the same amount of money 14 times more reads, but only 1/7 of writes compared to disks.

This evaluation revealed even for a DBMS a large optimization potential when its processing characteristics can be adjusted in a way that the drawbacks are avoided and, at the same time, the strengths of flash is exploited as far as possible. Here, we want to explore the energy-saving potential of DBMSs, in particular, when data caching is used together with flash as permanent storage.

3 DB CACHE MANAGEMENT

Caching plays a dominant role in all DB-based applications and its importance steadily grows with the number of Internet applications. In current DBMSs, the overall goal of cache management is to exploit locality of reference as far as possible thereby minimizing IO to the permanent DB on disk. When a requested page is not found in the cache, a *page fault* occurs. If a free frame is found in the cache, this page can be directly loaded from disk. Otherwise, a page replacement algorithm determines a "victim" to make room for the request. If the victim page is marked as updated, loading has to be deferred until this page is written (flushed) to the disk. To reduce such wait situations, the cache manager often gives such modified pages a preferential treatment and flushes them asynchronously (without being evicted) to make them again "clean" in the cache.

Most of the prominent page replacement algorithms, e.g., LRD (Effelsberg and Härder, 1984) or LRU-K (O'Neil et al., 1993), proven in disk-based DB applications, concentrate on single-page selections and are not flash-aware, because they do not consider the asymmetric cost of read and write or even the potential of multi-page fetches, e.g., an entire block content, which can be helpful as a kind of prefetching for specific workloads.

With flash, single-page flush optimization or victim determination is contra-productive. As a general principle, output should be absolutely minimized, potentially at the expense of input. Because writes need special handling and are, therefore, much more expensive than reads, page replacement should bias the eviction of read-only pages over modified pages which should be collected, before they are flushed. Ideally such a collection should fill a block on flash.

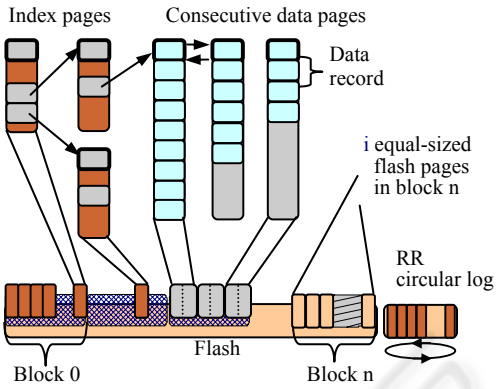

Figure 1: Logical-to-physical data mapping.

If there is no restriction on page types to be collected for flash mapping, a single cache is sufficient. However, if the flash is organized in type-specific files, this can be best achieved by type-specific caches controlled by a multi-cache manager, which only flushes complete blocks. To isolate update propagation from logging and recovery concerns, e.g., transaction-specific force-writes at commit, a *no-force* strategy is mandatory for modified pages (Gray and Reuter, 1993). A *no-steal* policy for dirty pages, i.e., pages modified by still running transactions, is not required, as long as the WAL principle is observed. Note, there is an important dependency between data cache flushes and log buffer writes (see Section 4.2).

Preliminary results show that flash-aware replacement using a single cache cannot support most workloads well. For scenarios sketched in Section 4, multi-cache management may outperform a single common LRU- or LRD-based cache by a substantial margin (e.g., more than a factor of 2).

4 MAPPING SCENARIOS

Flash-aware cache algorithms must address the IO asymmetry when performing writes and reads. First, let's look at the physical mapping of flash pages. Figure 1 illustrates that the flash can be divided into several files whose blocks may be either randomly written (e.g., for user and index data) or filled in a circular fashion (e.g., for log data). Consecutive data pages are often mapped to the same block whereas index pages are typically dispersed across several blocks. As an additional degree of freedom, the DB page size can be chosen as a multiple of the flash page size. Thus, a DB page is mapped to one or more consecutive flash pages, which, in turn, sets two major goals for DB page writing:

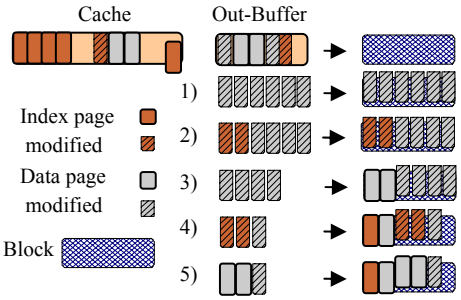

Figure 2: Out-buffering of pages.

1. Avoid expensive erasure by using always empty blocks for page writing.
2. Flush collections of modified pages such that new blocks are filled without fragmentation.

To achieve the first goal, writing single or few pages (chained IO) always into a new block causes tremendous fragmentation in this block. At the end, most of the blocks are poorly filled, read performance degrades, and flash capacity is wasted. Therefore, goal 2 prefers to buffer DB pages for output (out-buffer, for short) to fill flash blocks completely. Indeed, this avoids fragmentation as long as enough empty blocks are available to propagate new or modified data. But pages being relocated into new blocks leave gaps in their old blocks causing fragmentation and later garbage collection. Hence, replacement must balance both extremes to minimize overall writing.

Due to different DB page types for indexes, data (often chained), log, and temporary data (used for external sort and intermediate results), the propagation strategy decides whether it is better to assign to a block (rather) static together with (highly) dynamic pages or to separate them. From an application point of view, the context of related user data should not be destroyed due to page update propagation to preserve locality of data reference. On the other hand, maintaining the original block assignment of data may be overly expensive due to read/write asymmetry. To achieve reasonable page mapping with minimum fragmentation and maximal locality support, different out-buffering strategies (block-building) are possible.

4.1 Propagation Strategies

Figure 2 sketches strategies for out-buffering modified DB pages before flushing them to blocks.

- *In-place:* Substituting each page in-place avoids fragmentation at all, but requires each time the entire flash block to be erased and rewritten with all existing data.

Table 1: Comparison for propagation strategies.

Strategy	single page writ e	multiple page write (same type)	multiple page write (mixed types)
in place	- | - | +	- | - | +	- | - | +
relocate	o | o | +	o | o | +	o | o | +
new block	+ | + | -	+ | + | o	o | + | o
performance | power consumption | fragmentation - poor o average + good			

- *Relocate:* Shifting a page to a new physical position leaves a gap (outdated page) in the present block, but the page can be out-buffered together with other pages (even unmodified pages) to be written into another block.
- *Allocate new block:* When an empty block is used to be filled with new or modified pages, the block can be written by chained IO which is much more efficient than separate page flushes.

To summarize these ideas, Table 1 shows the anticipated performance, power consumption, and degree of fragmentation for single-page and out-buffered- page flushes. The combination of different page types to build blocks is disadvantageous for sequential data being spread over more blocks than necessary when being read later. Table 1 also reveals that the straightforward and easy to use *in-place* strategy does not perform well in most of the considered aspects. Depending on DBMS requirements, different propagation strategies favor different design goals (e.g., power, time, cost, space); cost-based decisions may choose the best strategy even on demand.

4.2 Mapping Data to Flash

So far, we have characterized the general properties and principles of mapping cached data to flash. For that purpose, Gal and Toledo (2005) already presented lower-level algorithms. To become more DB-specific in our position paper, we propose in the following how the mapping can be applied to important DBMS processing situations and workloads:

Log data: The properties of collecting and writing log data lend themselves perfectly to flash support. The log file is organized in a circular way and new log data is always appended to the current end of the log file which makes the log tail to a high-traffic data element. Because flash blocks are rather large, they normally are not filled by the log data of

a single transaction. However, repeated log writes of consecutively committing transactions to the same block would cause a performance-critical bottleneck. Therefore, a pool of block-size log buffers should be allocated, which are alternately used to enable filling of log data while a full block (using chained IO) is asynchronously written to flash. For this purpose, the concept of *group commit* was developed for which the log data are written in commit sequence (at least the end-of-transaction entries) to the log. The successful commit is acknowledged to the user together with the release of all locks kept by the committed transactions when the log data has reached stable storage (e.g., flash). If the delayed lock release is a performance problem, group commit can be combined with *pre-commit* (see Gray and Reuter, 1993) which immediately releases locks at the end of the individual transaction T_i although the log data is still in a volatile state. The only situation that such a transaction may still fail is a crash. But then all pre-committed (and other running) transactions potentially dependent on the updates of this transaction (because of early lock release) also fail because their log data appear after the log data of T_i.

Static user data: DB pages containing user data are primarily read and rarely updated, hence, almost static. In many cases, the physical allocation of DB pages on external media is of particular importance, e.g., if a cluster property has to be preserved. For that reason, data pages should be updated *in-place* as in disk-based systems, i.e., n single-page updates require n block writes. Note, wear leveling does not affect the cluster property at the block level, because random page reads are not sensitive to physical neighbourhood of blocks. Hence, management of DB pages within a block provides new opportunities to reallocate data on flash without loosing performance.

Append-on data: User data is often appended to the end of a file by using sequential numbering scheme for primary keys or to ordered XML trees which are right-growing when new order-dependent data is attached. In this case, the usage of a pool of block-size buffers, similar to that for log data, can dramatically improve the performance.

Dynamic user data: In such a scenario, page updates are assumed to be so frequent that block update in-place is too expensive because of a substantial number of single-page writes and, in turn, energy consumption. Therefore, we sacrifice the cluster property (automatically preserved for in-place block updates by the flash) and propose a

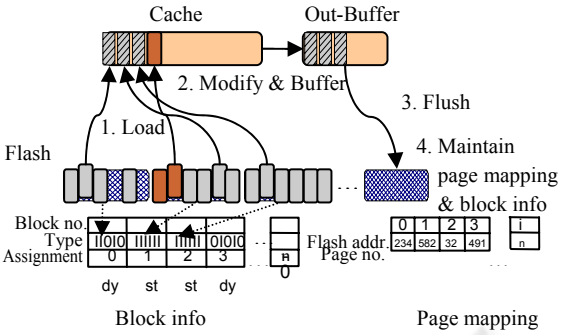

Figure 3: DBMS-controlled mapping to flash.

DBMS-controlled relocation, as illustrated in Figure 3. Modified pages are out-buffered until an entire block can be written. Administration information in the form of block info and page mapping is maintained by the DBMS and remains in memory. To enable recovery from a crash, the DBMS saves sufficient redundancy in a log file. Hence, if a flash block can store d DB pages, DBMS-controlled relocation reduces the write overhead of n pages to n/d block writes.

Index data: Index maintenance provides similar update frequencies as dynamic user data and, therefore, we propose a similar DBMS-controlled mapping (Figure 3). A first prototype of flash-aware index algorithms was developed by Nath and Kansal 2007, where memory had to cope with severe size restrictions. Although they omitted scalability considerations, they emphasized the need for flexible index maintenance to reduce expensive write operations. Due to the frequent but small changes within an index node, it is better to collect corresponding log information for that node update instead of flushing every single modification separately. Moreover, index traversals accessing single dispersed pages profit from the random-read speed of the flash. Other B*-tree operations, such as merge and split, may be deferred or covered by the same logging mechanism.

Hot spots: A critical task is the management of data which is updated with a high repetition rate (hot-spot data). The use of NAND flash implies the need of an adjusted approach. Due to their high locality of reference, the cache manager would never evict them for replacement in case of a no-force policy. The normal logging component is used to collect the needed redundancy to guarantee the ACID properties in case of a crash. To enable reuse of the log space, hot-spot data must eventually reach the stable storage. Therefore, a checkpointing

mechanism should be applied to periodically flush hot-spot pages using the normal out-buffer method.

4.3 Further Optimization Options

The forms of mapping and their effectivity/energy-saving potential discussed so far could be further enhanced by a number of optional methods.

Workload analysis: To optimize the read/write ratio, the cache manager may predict access patterns or pro-actively tolerate performance degradations to gain additional benefit for later read operations. For example, flushing sequentially referenced pages immediately may increase current write costs. But, because re-reference probability is very low such that the cache space freed may amortize the initial writing costs. Besides reducing gaps in existing blocks, a suitable propagation strategy compares write and read operations within a workload for a specific page type or each page individually. Hence, the cache manager has to distinguish between page types and their future usage.

Database design: Defining a flash-aware DB schema should reduce index usage or redundant data structures, because updates would affect too many pages. A higher amount of data to read, even for selective accesses or joins, may compensate the otherwise unacceptable update costs for such redundant data.

Replacement urgency: Regarding power consumption and latency for write compared to a read operation, the cache may prefer to eagerly drop non-modified pages. Indeed, modified pages may benefit from a deferred flush, as long as WAL is used and data consistency is assured. Thus, a page may be replaced containing modifications from several transactions resulting in one instead of multiple block writes (no-force policy).

5 SUMMARY AND OUTLOOK

We outlined the specific properties of NAND flash when used as persistent devices in DB servers. Although magnetic disks with salient and proven performance and reliability properties and, above all, acceptable capacities (512GByte and larger) will be available on the market, flash disks may provide even superior properties in the near future. The dramatic reduction of energy consumption and the potential to read random data nearly as fast as sequential data are outstanding advantages which make NAND flash memory to an almost perfect hard disk alternative. Because of the problematic erase operation of flash blocks, DBMS algorithms and, above all, the mapping between cache memory and flash disk have to be adjusted to reach the best performance possible. We illustrated the opportunities and draw-backs of various mapping scenarios and developed ideas for optimization. Primarily, the collection of data pages in block-size memory buffers may greatly improve the performance.

Currently, we are working on the integration of flash memory into a DBMS and its adaptation providing a flash-aware cache together with enhanced mapping algorithms. Other hot topics for the improvement of energy efficiency in DBMSs is the use of specialized logging techniques and group commit as well as cost-based query optimization regarding energy consumption as a prime cost factor.

REFERENCES

Ban, A., 2004. *Wear leveling of static areas in flash memory*. US patent, (6732221); Assigned to M-Systems

Effelsberg, W., and Härder, T., 1984. Principles of Database Buffer Management. In *ACM Transactions on Database Systems* 9(4): 560-595

Gal, E., and Toledo, S., 2005. Algorithms and data structures for flash memories. In *Computing Surveys* 37(2): 138-163

Gray, J., and Fitzgerald, B., 2007. *FLASH Disk Opportunity for Server-Applications*. http://research .microsoft.com/ ~Gray/papers/FlashDiskPublic.doc

Gray, J., and Reuter, A., 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann

Härder, T., and Reuter, A., 1983. Principles of Transaction-Oriented Database Recovery. In *Computing Surveys* 15(4): 287-317

Hwang, C., 2006. *Chip memory to keep doubling annually for 10 years: Hwang*. http://www.korea.net/ News/news/LangView.asp?serial_no=20060526020& lang_no=5&part= 106&SearchDay= (2006)

Kerekes, Z., 2007. (editor of STORAGEsearch.com): *SSD Myths and Legends – "write endurance".* http://www.storagesearch.com/ssdmyths-endurance.html

Nath, S., and Kansal, A., 2007. *FlashDB: Dynamic Self-tuning Database for NAND Flash*. ftp.research.microsoft.com/pub/tr/TR-2006-168.pdf

O'Neil, E. J., O'Neil, P. E., and Weikum, G.: The LRU-K Page replacement algorithm for database disk buffering. *Proc. ACM SIGMOD,* 297-306 (1993)

SAMSUNG Develops, 2008. MLC-based 128 Gigabyte, SATA II Solid State Drive. http://www.samsung.com/ global/ business/semiconductor/newsView.do?news_ id=893