Towards Flash Disk Use in Databases— Keeping Performance While Saving Energy?

Theo Härder, Karsten Schmidt,
Yi Ou, and Sebastian Bächle
Databases and Information Systems Group
Department of Computer Science
University of Kaiserslautern
D-67653 Kaiserslautern, Germany
{haerder,kschmidt,ou,baechle}@cs.uni-kl.de

Abstract: Green computing or energy saving when processing information is primarily considered a task of processor development. We, however, advocate that a holistic approach is necessary to reduce power consumption to a minimum. We explore the potential of NAND flash memory in comparison to magnetic disks for DBMS-based architectures. For this reason, we attempt to identify the IO performance of both storage types at the device level and file system interface, before we approach the most important question how DBMS applications can take advantage of the flash performance characteristics. Furthermore, based on an analysis of storage structures and related benchmark runs in our *XTC* system, we discuss how IO-intensive DBMS algorithms have to be adjusted to take most out of the flash technology which is entering the server area with dramatic pace. The ultimate goal is to substantially improve energy efficiency while comparable performance as in disk-based DB servers 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 primarily addresses efforts to reduce power usage, heat transmission, and, in turn, cooling needs of hardware devices. Thus, research and development primarily focus on 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 chips areas, but also combined efforts regarding system architecture (e.g., energy-efficient storage) and system software (e.g., energy-optimal algorithms). Moreover, virtualization of all system resources enables an energy-aware system management, e.g., by launching a control center which monitors workload changes and may respond by allocating/deactivation entire server machines. Even supply chains are considered from this perspective to minimize their "carbon footprint"[1].

Up to now, research has hardly contributed to this important overall goal in the database management system (DBMS) domain. But, NAND flash memory (also called solid state

[1] "SAP optimizes CO2 balance of supply chains", Computer Zeitung, Sept. 2008

disk or flash, for short) has the potential to become the future store for permanent database data [4]; it promises—compared to magnetic disks (disk, for short)—breakthroughs in bandwidth (IOps), energy saving, reliability, and volumetric capacity [6].

So far, flash storage was considered ideal for keeping 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 specific single-user workloads known in advance.

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 possibly take over the role of disks in server environments. However, not all aspects important for single-use devices count for DB servers and not all out-of-the-box DBMS algorithms can be directly applied and guarantee adequate performance when processing data stored on flash.

In this paper, we want to explore whether flash disks—from a performance point of view—can take over the role of or even replace magnetic disks in DB server environments. In Section 2, we summarize the potential of flash storage as a competitor of disks in such environments. While decades ago the device-related IO performance completely determined the cost of application IOs, nowadays a variety of built-in optimization mechanisms in the IO path of DBMS applications may relieve the real IO overhead and improve the related IO cost suchlike that the plain device characteristics may only play a minor role. For this reason, we evaluate in Section 3 the device-related IO performance of both competitors, where the influence of the OS file system is avoided as far as possible. In Section 4, more detailed IO performance tests are conducted at the file level to identify the influence of the OS and the file system. To figure out the discrepancy of flash and disk efficiency at the DBMS level in Section 5, we outline three native storage mappings of XML documents available in *XTC*, our native XML DBMS [12]. These storage mappings serve to discriminate the number of IO requests needed when processing XML workloads and help to reveal in Section 6 the DBMS-related IO performance for the storage devices compared. Finally, we throw a glance at the question of energy saving, before we conclude our findings in Section 7.

2 Flash Usage in DB Servers?

Usually, fixed-size *blocks* are mapped to external storage and are the units of IO (data transfer of a byte sequence) from an external device to the OS. Such blocks, in turn, are stored on disk as a consecutive sequence of smaller units, called segments typically consisting of 512 B. Storage representation is organized differently on flash. The entire storage space is divided into m equal *flash blocks* typically much larger than disk blocks. A flash block normally contains k (32–128) fixed-size *flash segments*, where a segment ranges between 512 B and 2 KB. Because of the NAND logic, an individual segment cannot be directly updated, but needs *erasure* (reset) of the flash block it is contained in (automatically done by the flash device), before it is rewritten. The flash segment is the smallest and the flash block the largest unit of read, whereas the flash segment is the unit of write; using chained

IO, the DBMS, however, can write $1<i\leq k$ flash segments into a block at a time. Note, whenever a flash segment is written in-place, *wear leveling* [1] automatically allocates a new flash block and moves to it all segments from the old flash block together with the updated segment (keeping an existing cluster property). Restricted *write endurance*, referring to the maximum possible erase cycles—between 100,000 (older references) and 5,000,000 (most recent references)—, would contradict general server usage. However, plenty of mapping tricks are provided [16] or it even seems to be a non-problem[2].

2.1 Read/Write Models

Disks are devices enabling fast sequential block reads and, at the same time, equally fast writes, whereas random block read/writes are much slower (requiring substantial "mechanical time"). The block size can be configured to the needs of the DBMS application with page sizes typically ranging between 4 KB and 64 KB. To hide the access gap between memory and disk, a DBMS manages a large DB cache in memory (RAM) where each cache frame keeps a DB page which, in turn, is mapped to a disk block. In most DBMSs, page propagation uses update-in-place applying WAL [8].

2.2 Flash vs Disk Performance

In the literature, there is a unanimous agreement on the following facts [4, 6, 15]: Disks are excellent *sequential* stores, whereas random access is only improving for larger page sizes. On the other hand, flash offers generally better sequential/random read (SR/RR) and comparable sequential write (SW), but terrible random write (RW) performance:

- SR and SW having a bandwidth of more than 60 MBps for page sizes $\leq$ 128 KB are superior, but remain comparable to those on fast disks.
- RR is spectacularly faster for typical DB page sizes (4–32 KB) by a factor of 10–15 ($\geq$3000 to $<$200 IOps).
- RW performs worst with $\sim$50 IOps, opening a huge advantage for RR, and is slower by a factor of 2–4 compared to disks (for the page size range considered).

Technology forecast is even more optimistic: J. Gray [7] claimed a flash read bandwidth of up to 5000 random reads per second or an equivalent of $\sim$ 200 μsec per 8KB page. Furthermore, an internal Microsoft report [6] predicts that about 1,100 random 8KB writes per second seem possible with some flash disk re-engineering.

This picture is complemented by the energy consumption: The power needed to drive a flash read/write is $\sim$ 0.9 Watt (in our case) 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 or 55 flash-writes can be achieved per Watt, whereas a disk only reaches 13 operations per Watt ([6], [18]).

[2] A worst-case thought experiment [14] with repetitive sequential block overwrites on a flash indicates that 51 years are needed to reach the wear leveling limit.

For more than a decade, NAND flash chips doubled their densities each year and currently provide 64 Gbit. This growth will continue or accelerate such that 256 GB per chip are available in 2012 [13]. Because several of these chips can be packaged as a "disk", we should be prepared for flash drives with a capacity of several Terabytes in this near future. Of course, flash is still quite expensive. A GB of a flash device amounts to >3$ (but only <0.2$ for disk), but technology forecast predicts rapid market growth for it and, in turn, a dramatic decrease in the near future that it will roughly reach the disk[3] [5, 15].

3 Device-Related IO Performance

In contrast to impressive flash behavior, performance gains to be anticipated when integrating flash devices into a DBMS are not easy to obtain. After a long period of experiments under DBMS control—painful because of results which could not be interpreted at all—, we started from the scratch to explore step-by-step the influences of built-in optimization mechanisms in the IO paths of blocks read from or written to external storage.

3.1 Historical Note

40 years ago, timings of disk access from the file interface was more or less (very close to) those of raw-disk access, because all block transfers from the plain disk were directly done via a channel to some address space in RAM. There was no disk cache available and further abstractions (such as file buffers) achieved by the OS file system were not present. Therefore, it was rather simple to compute disk access times for various kinds of block transfers. Technology, device, and usage parameters, provided by the disk manufacturer, were sufficient to calculate for given access requests the expected block access times. For this purpose, the disk access time t_d of a block of size b, where x specified the number of cylinders to be crossed from the current access arm position to the cylinder carrying the block requested, was computed by

$$t_d(x,b) = t_{SIO} + t_s(x) + t_r + t_{tr}(b).$$

Here, t_{SIO} is typically in the range of μsec and stands for the CPU overhead (OS supervisor call, device activation, etc.) until device-related activities can begin. The lion share of the access time was the disk seek time (access motion time) $t_s(x)$, which was specified by the manufacturer as a non-linear increasing function. Furthermore, the rotational delay and the data transfer were captured by the remaining variables.

To obtain a sufficiently accurate approximation, t_{SIO} could be skipped in practical cases. Moreover, the calculation of $t_s(x)$ can be facilitated by approximating the course of the seek time using linearization in sections. The specific distances x (for sequences of consecutive requests) could be derived from the file-to-cylinder allocation and the application requests. To enable access time calculation for a particular sequentially allocated file, certain device constants are given for each disk type: $t_{s_{min}}$ (for $x = 1$) and $t_{s_{max}}$ (for $x = N_{dev}$).

[3] We observed for a 64GB flash a dramatic reduction of $\sim$ 70% within 90 days down to $\sim$ 3.5 $/GB, whereas a 750 GB SATA only reached $\sim$ 7%/GB down to $\sim$ 0.15 $/GB.

$t_{s_{avg}}(x)$ assumes random access to blocks uniformly distributed over N consecutive cylinders of the device for which $x = N/3$ is obtained as the average distance. The expected rotational delay t_r was half a revolution ($t_{rev}/2$) of the disk platter. And, finally, the disk transfer rate F_d was dependent on the track capacity T_{cap} in KB (and, in turn, bit density) and revolution speed (e.g., R = 7,200 rpm), which resulted in $t_{tr} = b/(T_{cap} * R) = b/F_d$ sec for a block size of b in KB. An often used approximation formula [9] for t_d was

$$t_d(x,b) \approx t_{s_{avg}}(x) + t_r + b/F_d.$$

Because the avg. travel distance of the disk arm is hard (or even impossible) to determine in a multi-user environment, the avg. seek value—specified as $t_{s_{avg}} = t_s(N_{dev}/3)$ by the manufacturer—was used instead. Hence,

$$t_d(b) \approx t_{s_{avg}} + t_{rev}/2 + b/F_d$$

where besides the application-specific variable b only device characteristics occur.[4]

3.2 Device-related Experiments

Today, such simple access scenarios do not exist anymore for disks, because many optimization features were added along the IO path from disk to the application address space in RAM. A typical mechanism at the disk level is the existence of a (RAM) cache—in our case 8 MB—which cannot be controlled by the application. In fact, depending on the current workload, it automatically adjusts the sizes reserved for read and write caching.

Flash memory is now available for more than 20 years. So far, it was not considered as a serious competitor for the disk because of size and cost. However, these aspects have dramatically changed in recent years. Because flash consists of electronic memory, "mechanical movements" do not stretch the device access times, which are therefore much simpler to calculate. Furthermore, a flash is not equipped with a volatile cache, which would speed-up IO while significantly consuming more energy due to the permanent refresh cycles. However, the caching and prefetching options for the OS file system remain the same as for disks. Hence, if we could minimize the file caching effects as far as possible, we should be able to approximate raw-flash timings in a controlled measurement experiment. Because "mechanical time" is not present, when a block is read, flash access time $t_f(b)$ for a block of size b (b given in KB and independent of its location) can be approximated by

$$t_f(b) \approx t_{SIO} + b/F_f$$

where t_{SIO} is roughly the same as for disks. But, it cannot be neglected here because of its relative magnitude compared to the data transfer time b/F_f from the flash device.

To gain a deeper and more complete comparative picture of flash and magnetic disk, we performed dedicated throughput tests. While sequential read/write performance of flash

[4] For example, disk IBM 3330 rotating at 3,600 rpm was characterized by $t_{s_{avg}} = 30$ ms, $t_{rev} = 16.7$ ms and track capacity of 13 KB, resulting in F_d = 690 KB/sec, which enabled the calculation of the avg. block access time with sufficient precision, e.g., $\sim$ 43.9 ms for a 4KB block.

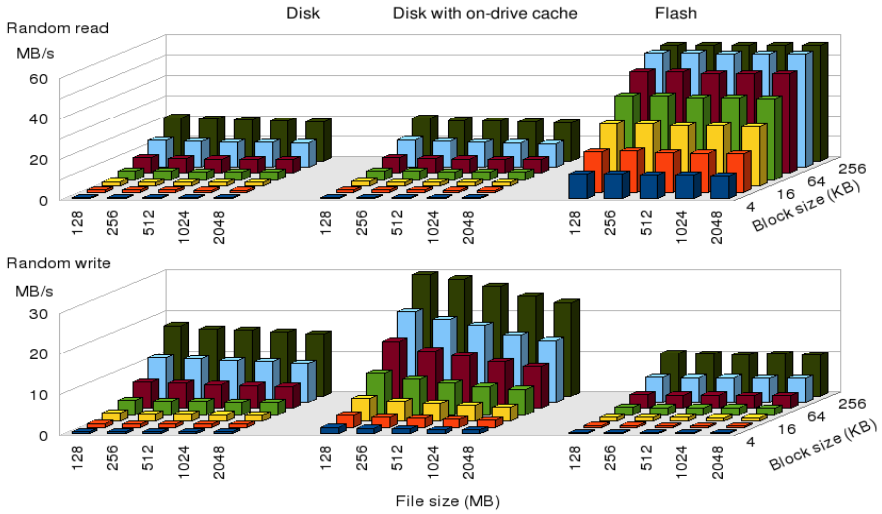


Figure 1: Flash and disk throughput for random reads/writes

and disk is comparable, we focused on random read/write operations at the device level where flash promises the biggest gain for reads, but also the strongest penalty for writes.

Figure 1 summarizes the throughput results (MB/s) of our tests at the LINUX file interface (using iozone[5]), where the OS caching effects were eliminated (to the extent possible) and no buffer pool was explicitly allocated. We varied the file size from 128 MB to 2 GB and the block size as the unit of access from 4 KB to 256 KB. All experiments started with a cold RAM and cold disk cache. In all cases, dividing the file size by the block size delivered a range of integers where *random numbers* for block reads/writes were taken from. The number of accesses were computed by *file size/block size* which guaranteed that (almost) the entire file content was fetched to or written from RAM. Our performance figures are gained for mid-range devices (SSD [21] and SATA 7,200 rpm). Although many design parameters can be optimized [6], the relative IO performance characteristics to be anticipated from both device types remain stable and are also valid for high-end devices. Therefore, the results contained in Figure 1 and the cross-comparisons can be considered indicative for the device-related throughput performance.

Referring to the measured results for flash *random read* in Figure 1, we can derive the values for the system-specific t_{SIO} and the device-specific F_f (65 MB/sec sequential read rate) to obtain an analytic approximation by

$$t_f(b) \approx (230 + b/0.065)\mu sec.$$

Hence, a 4KB block can be read from flash in < 300 μsec whereas read/write access to disk needs $t_d < 10$ ms for a 4KB block. Here this is due to the ratio of file size (≤ 2 GB) and disk size (80 GB) where frequent track-to-track seeks accounted only for 2 ms and an avg. latency of 4.2 ms.[6] As expected, a flash write with 4 KB costs ≥ 20 ms.

[5] http://www.iozone.org

[6] An experiment with 80GB file size would consume $t_{s_{avg}} = 15$ ms for the disk and result in $t_d \approx 20$ ms.

While the overall IO performance of flash and plain disk confirm our statements in Section 2 and correspond to our expectations, closer inspection of Figure 1, however, reveals that, even for the device-related IO performance, cross-comparison and performance assessment in detail is not that easy. For example, the slight decrease of the read throughput for both device types with growing file size is not obvious. This effect is caused by the indirect addressing which provokes some additional block transfers for address information (see Section 4). Furthermore, larger files imply longer arm movements for the disk, however, in our case, to very limited degree.

The access behavior of disks with on-drive cache may substantially deviate from the plain case. Because we started with cold caches, *random read* could not profit. But, *sequential write* or *random write* throughput could be considerably improved, because blocks are buffered in the 8MB cache and asynchronously propagated to disk. Although we did not explicitly measure *sequential* IO performance, it is approached for *random read* ($\sim$ 65 MB/s) and *random write* ($\sim$ 10 MB/s) on flash using large block sizes (e.g., 256 KB). As per device specification, *sequential read/write* disk IO delivers $\sim$ 60 MB/s.

4 File-System-Related IO Performance

The file system embodies a major part of the IO path and presents a substantial deviation when application programs such as a DBMS invoke IO requests. Thus, a certain overhead and fuzziness—as far as interpretation of the IO cost is concerned—is induced by the optimization options of the OS kernel (i.e., IO clustering, prefetching, and kernel-related caches). To assess flash and disk access, we need to figure out how the additional abstraction level of the file system affects reachability and appearance of the underlying storage device for the program. Therefore, we try to disclose the cache hierarchy thread.

All our benchmarks and experiments were performed on an AMD Athlon X2 4450e (2.3 GHz, 1MB L2 cache) processor using 1 GB of main memory (RAM) and a separate disk for the operating system. As hard disk, we use a WD800AAJS (Western Digital) having an 8MB Cache, NCQ, and 7,200 rpm. The flash device is a DuraDrive AT series (SuperTalent) having a capacity of 32 GB. The operating system is a minimal Ubuntu 8.04 installation using kernel version 2.6.24 and the Java Virtual Machine is version 1.6.0_06.

Because all layers in the cache hierarchy have well-known interfaces, they may be separately substituted, but normally they are not elided. Figure 2 illustrates the OS- and device-controlled caches. As described in Section 3.2, disks benefit from integrated caches in case of sequential reads/writes or random writes, as opposed to flash devices which do not have such a cache. At the lower OS kernel interface, a *kernel cache* is allocated that allows for buffering entire blocks (as delivered by the device) using an LRU-based propagation strategy. It is automatically resized depending on available main memory. Cache filling is optimized for sequential accesses by prefetching. In the layer above, a virtual file system (VFS) maintains several caches for *inodes*, *directories*, and (application-formatted) *pages*. As these caches are solely controlled by the kernel, flushing them is the only functionality provided for the application. Typically, page cache and kernel cache hold actual, but not explicitly and currently requested data in RAM. In contrast, the *inode cache* and *directory cache* speed-up file resolution, block addressing, and navigation.

4.1 Caching Hierarchy

At the OS level, room for file caching may be reserved in such a way that all RAM currently not needed by the OS and the applications is used for caching block reads and writes. The file system may use this cache to prefetch blocks from disk in a way not visible and controllable by the applications. This kind of prefetching is often heuristically performed depending on the locality or sequentiality of current block accesses to disk. For example, Linux offers the following options:

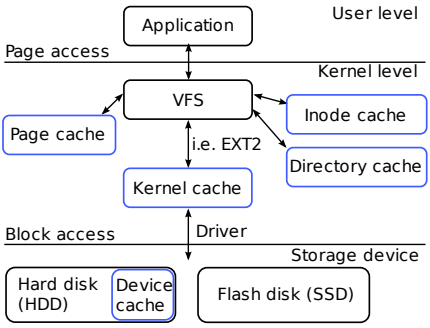


Figure 2: Caching hierarchy

- The first file access conservatively decides on prefetching. Only when the *first block* is read, sequential access is anticipated and some minimal prefetching takes place.
- Synchronous prefetching can be enabled to lower seek costs for larger reads.
- Read access pattern recognition may dynamically trigger synchronous and/or asynchronous prefetching.

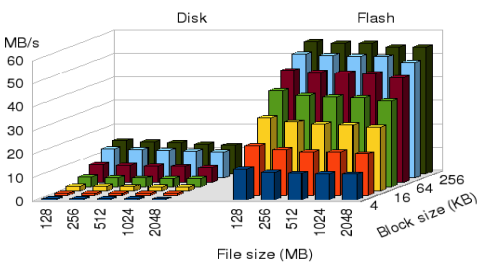
To enable greater flexibility, storage access optimization is further separated into raw-block caching and file-structure caching (for inodes, indirect block references, etc.). Moreover, some access options (direct, sync) are provided at the file system interface. However, not all OS versions (in our case Linux) observe such application directives, e.g., even if the raw-disk option is enforced, the OS does not respond to it. As a result, great indeterminism may be caused for all kinds of disk IO timings in real application environments.

4.2 File Structure Mapping

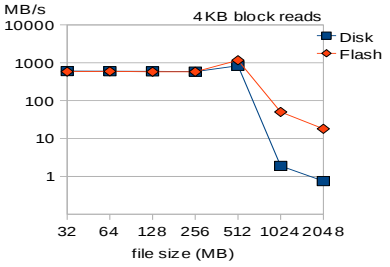
In Linux systems, the dominating file systems are EXT2 and its journal-enabled version EXT3, respectively. EXT2/3 divides the disk partitions in equal-length groups containing descriptors and bitmaps for inodes and block allocation. An inode is a special data structure describing a file's size, length, block allocation, etc. Because an inode only references 12 data blocks directly, the remaining data blocks for files larger than 12 blocks need to be referenced indirectly. This indirection of one, two, or even three levels can lead to an overhead of one-tenth of a percent for large files. Certain access patterns may suffer from such "long" block reads when several indirect block references have to be resolved before.

4.3 File/Device Caching Effects

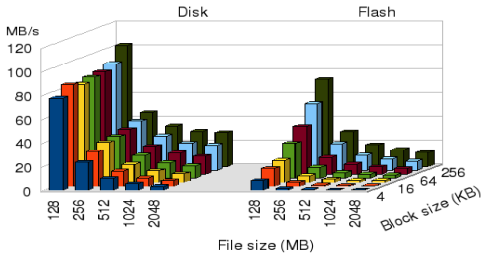
Prior work in [3] has shown that caching effects may closely interfere with the kernel's prefetching and replacement strategy. In all our experiments, we used the default LRU replacement and the default prefetching of the Linux kernel, because we want to compare



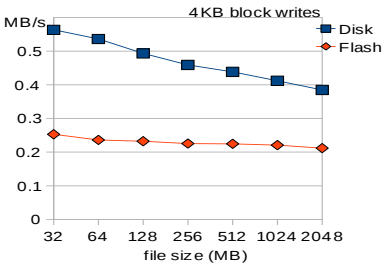
(a) Random read, cold buffer



(b) Hot buffer effect



(c) Random write, async



(d) Sync effect

Figure 3: File system benchmarks and their effects: cold vs. hot buffers for random reads (a vs. b) and unforced vs. forced random writes (c vs. d)

their effects for the different types of physical devices. Furthermore, we constrained some of our experiments to the default block size for the file system of 4 KB to limit the result dimension to a reasonable degree. Of course, when perfect knowledge of future access is available, the file-system block size can be adjusted to the minimal applications' page size, thereby gaining the most from caching and prefetching. However, real systems are faced with varying access patterns and the 4KB block size seems to be the lowest common unit.

Comprehensive read and write benchmarks, as shown in Figure 3, have revealed that the OS' page cache and the synchronous-write option may seriously affect the IO behavior of the underlying devices in different ways. Note that we illustrate our experiments in Figure 3a and Figure 3c only for file sizes of $\geq$128 MB, because the results for smaller files were dominated by caching effects that even cold-buffer experiments could not be explained in a plausible way. Figure 3a reveals the slight degradation as compared to that of Figure 1 due to the additional algorithmic overhead of the kernel's optimization efforts. In contrast, Figure 3c clearly shows the additional caching effects, when data is written asynchronously (default case). For smaller amounts of data (less than main memory available), most of the data is only written to the cache and not propagated to the physical device, leading to high throughput rates. Having more data to write than the available memory can cache, the asynchronous flush daemon (pdflush) is noticeably slowing down the throughput into the raw device's performance region (see Figure 1).

Another interesting effect appears when the IO behavior of hot and cold buffers is compared. As real systems do rarely restart, the buffers are more or less hot during runtime. Figure 3b shows the effect when operating on 4KB blocks: once the dataset does not

fit into main memory, the throughput drastically drops to the cold buffer rates. It further reveals that the throughput for small datasets is completely decoupled from the underlying storage, because the disk and flash throughput rates are equal up to a file size of 512 MB.

Because our primary application is database-based, the distinction between synchronous and asynchronous write is essential. The default file system behavior, shown in Figure 3c, does not enforce immediate write propagation at all. When using the *sync flag* or calling the system function *sync*—mandatory for some specific writes within transactions to guarantee the ACID paradigm—substantial throughput reduction has to be taken into account (compare Figure 3d and Figure 3c for 4KB block writes). Figure 3d shows for 4KB block writes that the throughput rates are independent of the file size. The slight decrease is explained by the file structure overhead discussed in Section 4.2 and the additional seek overhead for large files in case of a disk.

The simple and often constant throughput rates of the raw devices are markedly affected by the file system and the kernel's caching efforts. Thus, an application (i.e., our database) needs to be aware of that when using access paths or building cost models.

4.4 Non-influenceable Caches and Other Magics

Keeping instructions and data close to the CPU(s) in the processor caches can significantly change the anticipated IO behavior. Whenever physical access is not needed at all, certain OS-internal optimizations are effective to improve (or at least to influence) the read/write throughput. In some situations, where definitely no physical IO occurred anymore, read throughput reached an extraordinary level. In Figure 3b, it is up to three orders of magnitude ($\sim$ 600 MB/s) higher for file sizes less than the available main memory. A peak (of $\geq$ 1000 MB/s) occurs in this measurement series when processing a file of 512 MB, before the throughput dramatically decreases. For this "magic" behavior which is independent of the flash/disk distinction, we found the following explanations:

- The CPU has small, but really fast processor caches (so-called L1 and L2). They typically range from KBs to several MBs, enough to keep frequently referenced data (sometimes entire files) as close to the CPU as possible avoiding MMU and IO access. Unfortunately, parallel processes on multiple cores or operating on GB-sized data volumes cannot exploit it.
- In the identified peak situation, the kernel cache does not need to be accessed anymore, because the requested pages reside in the page cache and the processor caches. In particular, the processor caches have reached a state where at least the addressing information is steadily available, which may explain why the throughput is almost doubling, although already at a high level.
- Cached entries (pages, blocks) have a state (e.g., clean, locked, dirty) and can be accessed in isolation or shared among several processes. Different states involve different access control operations leading to more or less computational overhead.

There are further "peculiar" situations which affect pure throughput tests. However, dealing with large files or data volumes (i.e., much larger than the available main memory—typically several GB or TB nowadays), all of them are nearly smoothed to negligibility.

5 IO-Relevant DBMS Processing Concepts

The IO characteristics of flash indicate that disk-based DB performance cannot be obtained in an easy way, not to mention by simple replacement. Substantial bandwidth gains are possible for random reads, while random writes are problematic and have to be algorithmically addressed at the DBMS side. If frequent writes are necessary, the adjustment of (logical) DB pages to the device's block sizes may relieve the performance bottleneck. In mixed read/write workloads, write caching will play a major role to realize write avoidance. Further, the potential of sequential writes should be consequently exploited. Hence, DBMS processing should be adapted such that the drawbacks are avoided and, at the same time, the strengths of flash are used to full capacity.

5.1 Prime Optimization Principles

In several respects, performance and energy efficiency in DBMSs are not conflicting design goals and should be jointly approached, where appropriate. It is always a good idea to optimize at a logical or algorithmic level and not to purely rely on device-based improvements. When using external storage, the most important design principles are:

- Design compact and fine-grained storage structures to minimize IO for allocating, fetching, and rewriting DB objects.
- Use processing concepts which maximize memory-resident operations and minimize random read/write access to external storage—in particular, avoid random writes to flash to the extent possible.

We discuss these principles for native XML DBMSs and outline various XML storage structures, which help to reveal critical aspects of DBMS-related IO performance. In general, saving IO is the major key to performance improvements and, in turn, energy efficiency in DBMSs. Thus, both goals are approached by using *compact* storage formats, because *plain* XML documents contain substantial redundancy in the structure part, i.e., the inner nodes and paths of the document tree (see Figure 4a)[7]. However, the tree structure should be preserved to support fine-grained document processing. Despite IO minimization, storage mapping should be flexible enough to enable dynamic modifications.

5.2 Structure Encodings

Natively storing XML documents requires identification of the resulting tree nodes, i.e., the assignment of node labels. Because performance, flexibility, and effectivity of many XDBMS-internal operations critically depend on them, the appropriate label scheme design is a key issue for the system implementation. Early implementations, primarily optimizing static XML documents, mostly voted for *range-based labeling* schemes. They

[7] The Computer Science Index *dblp* currently contains almost 10^6 instances for path class */bib/paper/author* and even more for element name *author*.

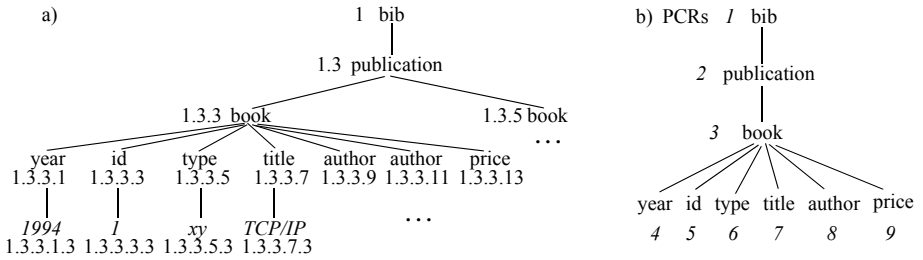


Figure 4: Labeled document (DOM tree) and its path synopsis

provide satisfactory query evaluation support: all axis relationships can be directly decided by comparing node labels. However, dynamic insertion of subtrees would cause a relabeling of the document (or a fraction of it) triggering a bulk IO operation. Moreover, they fail to provide the ancestor IDs when the ID of the context node is known, which is most important for setting intention locks when fine-grained multi-granularity locking is applied [12]. But, *prefix-based labeling*, derived from the concept of DeweyIDs, remains stable also in dynamic documents and enables all operations mentioned without document access—only by checking node labels [11]. Such labels as shown in Figure 4a support all optimization principles; we denote them SPLIDs (Stable Path Labeling IDentifiers).

A *path synopsis* represents equal path instances, i.e., those having the same sequence of element and/or attribute nodes, in an XML document only once as a so-called path class. PCRs (path class references) are added to the nodes of the path synopsis to identify the path classes, as illustrated in Figure 4b for a cut-out of the *dblp* document [17]. Typically, a small memory-resident data structure is sufficient to it during XML processing.

5.3 Structure Virtualization and Storage Mappings

Because of the typically huge repetition of element and attribute names, getting rid of the structure part in a lossless way helps to drastically save storage space and, in turn, document IO. As a consequence, log space and log IO may be greatly reduced, too. The combined use of SPLIDs as node labels and a path synopsis makes it possible to virtualize the entire structure part and to reconstruct it or selected paths completely on demand. When, e.g., SPLID=1.3.3.5.3 together with PCR=7 is delivered as a reference (e.g., from an index) for value *TCP/IP*, the entire path instance together with the individual labels of the ancestor nodes can be built: *bib/publication/book/title*.

B*-trees—made up by the document index and document container—and SPLIDs are the most valuable features of physical XML representation. B*-trees enable logarithmic access time under arbitrary scalability and their split mechanism takes care of storage management and dynamic reorganization. As illustrated in Figure 5, we provide an implementation based on B*-trees which cares about structural balancing and which maintains the nodes stored in variable-length format (SPLID+element/attribute (dark&white boxes) or SPLID+value (dark&grey boxes)) in document order. In the following, this XML storage mapping is denoted as *full* document representation.

As signified in Figure 4a, a SPLID is composed of so-called *divisions* and its length is dependent on the position of the corresponding node in the document. The value of a division identifying a node at the given level increases with the breadth of the document tree and, because it is inherited to the SPLIDs of all descendants, it affects the related descendent SPLIDs at all lower document levels. Although we use efficient Huffman encodings for the division values [11], they may reach substantial lengths in large documents. Note, the TPoX documents [19] consist of several million subtrees rooted at the second level in our 10GB experiments. Because the nodes in all storage mappings occur in document order (see Figure 5), their labels lend itself to prefix compression. Therefore, quite some storage space per document can be saved on external devices. On each page of the document container, only the first SPLID is completely stored. By storing the difference to the preceding SPLID and encoding the length of the SPLID prefix (in terms of divisions), which can be derived from the sequence of preceding node labels in a page, we obtain an effective compression mechanism (e.g., instead of storing 1.3.3.7.3, the encoded difference to the previous SPLID is enough). As a result, we gain the *prefix-compressed (pc)* storage mapping where in case of dense labels the avg. size needed for SPLID storage is reduced to $\sim$20–30 % [10].

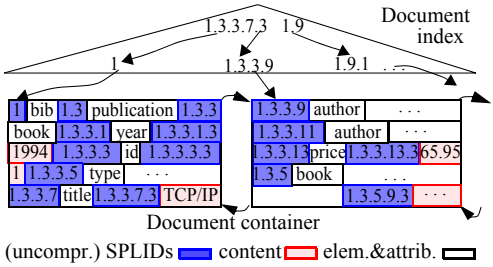


Figure 5: Storage mapping to a B*-tree

A virtualized structure only stores content (leaf) nodes in the container pages where each node is made up of a prefix-compressed SPLID, the PCR, and the text value. Hence, when storing or reconstructing the entire document or when accessing an individual root-to-leaf path, even more IO is saved as compared to the *pc* storage representation. We denote this kind of mapping as *elementless* XML storage mapping (*eless* for short).

In the following, we use all three storage mappings of XML documents in comparative measurements.[8] We applied the TPoX tools [19] to generate the various documents in the external text format, called the *plain* format, from which the differing storage mappings were created. To assess their storage effectivity, we cannot refer to absolute numbers, because the sizes of our sample documents vary from 10 MB to 10 GB. Therefore, we have normalized the space figures w.r.t. to *plain* (100%). Hence, the storage footprints of *full*, *pc*, and *eless* obtained quite some reduction and reached $\sim$95 %, $\sim$70 %, and $\sim$65 %, respectively; these figures directly characterize the IO saving when storing or reconstructing the entire document.

To later enable interpretion of query evaluation, we summarize in Table 1 the storage space needed for the TPoX-generated *plain* documents[9] when mapped to the *full* storage representation. For the corresponding footprints of *pc* and *eless*, we obtained roughly 75% and 70% of the pages numbers given. In particular, the growing sets of index pages and, in turn, the increasing heights of the document index h_{docind}, which together imply less buffer locality in our benchmarks, will affect the scalability results.

[8] Here, we do not consider content compression. It would further decrease IO overhead, but increase CPU time for additional compression/decompression tasks.

[9] We adjusted the TPoX generation parameters to roughly obtain *plain* documents of the given sizes.

Table 1: Footprints for *full* storage mapping of the TPoX documents.

Documents	Number of index / container pages (4 KB) [index height]			
	10 MB	100 MB	1 GB	10 GB
customer	7 / 967 [2]	78 / 9,317 [2]	964 / 121,026 [3]	10,794 / 1,219,099 [3]
order	7 / 871 [2]	87 / 10,650 [2]	728 / 101,641 [3]	9,501 / 1,118,463 [3]
security	7 / 982 [2]	85 / 10,493 [2]	220 / 40,385 [3]	3,382 / 407,604 [3]

6 DBMS-Related IO Performance

The flash characteristics highlighted in Section 2 promise quite some gain in performance and energy saving when considered in isolation. The interesting question is whether and to what degree their potential can be exploited in an XDBMS context? Furthermore, how can the optimization principles (see Section 5.1) assist our objectives?

A recent contribution [2] made us aware that "it is amazingly easy to get meaningless results when measuring a flash device because of the non-uniform nature of writes" even at the device level. For the interpretation of DBMS-related measurements, we have to cope with indeterminism provoked by uncontrollable OS caching and influence of DBMS buffer locality (which together fooled our initial complex measurements for quite some time). Therefore, we designed simple experiments which enabled meaningful result interpretation in the first place. Of course, they should reveal some (not all) important performance differences of flash- vs. disk-based DB systems.

To get an initial answer, we used *XTC* and implemented—by varying our options concerning external storage (disk and flash) and document representation (unoptimized (*full*), prefix-compressed (*pc*), and elementless, i.e., virtualized (*eless*))—six *configurations* of it. Furthermore, we designed a simple benchmark consisting of reader and writer transactions and ran it in single-user mode. Because we are solely interested in the relative performance behavior of the workload on the different system configurations, the processing details for the documents and transactions are only roughly sketched.

6.1 Workload Description

The performance analysis at the DB level is based on the well-known TPoX [22] benchmark to achieve a mixture of random/sequential access and a varying data distribution. Because most of the TPoX queries are quite complex and their detailed analysis would go beyond the scope of this paper, we chose the following queries supporting our objectives:

RQ1 report an account summary
RQ2 resolve a customer
RQ3 return a security price value
RQ4 return an order
RQ5 search for a specific security
WQ1 create a new account
WQ2 place an order
WQ3 update a price value

The *RQ** queries are read-only and the *WQ** queries include a share of updates or inserts. To reflect the indeterminism of real environments, all queries are supplied with

random parameters. By changing the weights of the queries, the share of the write load for the benchmarks is scaled from 10 % to 30 %.

Each benchmark run accesses the three XML documents *customer*, *order*, and *security* with almost uniform frequency. In all cases, a query selects some random target in a document with a given SPLID delivered from an element index—the only additional index used for these benchmarks. After having traversed the document index and located its context node, each query locally navigates in the document (see Figure 5) thereby accessing a number of records. Because the subtrees are small, these operations remain local (restricted to one or two container pages). Hence, the set of queries—each executed as a transaction—causes fine-grained and randomly distributed IOs on the XML database.

A major aspect of our analysis includes scalability properties. Each setup consisted of separate databases for the configurations *disk/full*, *disk/pc*, *disk/eless*, *flash/full*, *flash/pc*, and *flash/eless*. Choosing setups with 10 MB to 10 GB for the documents (in *plain* format), the system behavior is explored across three orders of magnitude. Therefore, the document footprint on a storage device (especially affecting the seek times on disk) and, in turn, the height of the document index considerably affect the length of a traversal path.

To gain some insight into the flash-disk dichotomy, we modeled two different processing situations where IO activities resp. complex computations in memory dominate the benchmark. Thus, we ran each benchmark in two versions. *IO-intensive* means that query evaluation query is the main transactional activity and that each transaction is started as soon as the preceding transaction committed. In contrast, in the *CPU-intensive* version we prolonged each transaction by 50 ms to account for complex main-memory computations.

6.2 Performance Measurements

There are literally infinite combinations of benchmarks, document configurations and other parameters. Here, we cannot provide area-wide and exhaustive empirical performance measurements. Instead, we intend to present some indicative results which focus on the cross-comparison of the differing configurations. For the different storage mappings and the considered devices, our measurement settings enable to highlight the influence of the OS- and device-related caches, to reveal some scalability aspects, and to figure out the flash benefits under different workload types.

The benchmarked workloads (consisting of 1K and 10K transactions[10]) were randomly assembled from the set of pre-defined queries and executed on the four DB sizes (setups). Within each setup consisting of 6 configurations, exactly the same queries were chosen, such that the differences in the IO behavior are only caused by the different storage mappings and the device type. To account for DB growth, only the query parameters were adjusted in the different setups to address document paths randomly spread over the entire document. While the IO activities on container pages remained rather unaffected by our scaling measures, the traversals through the document indexes were not. See Table 1 for the page numbers of the indexes, which determine their heights h_{docind}.

[10]Fewer transactions in a benchmark lead to lower *tps* rates, because they can—only to a lesser extent—take advantage of the increasing caching effects caused by index locality. But, on the other hand, their results are affected by singular and initialization effects. Therefore, such benchmark results are not reported here.

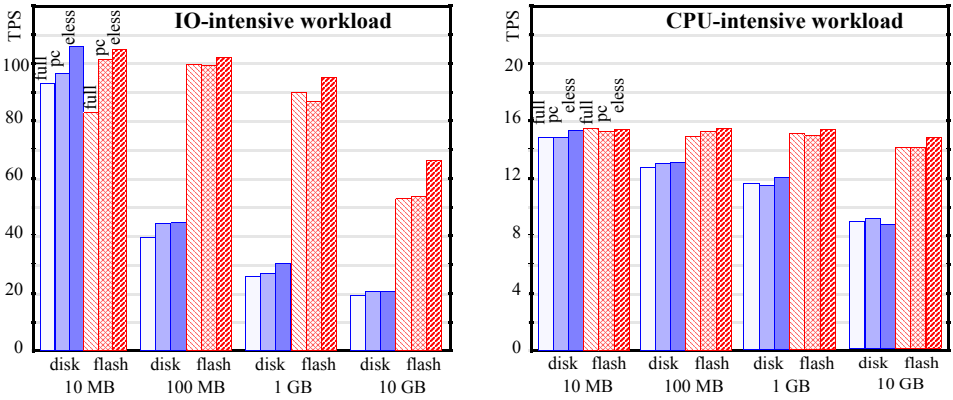


Figure 6: Reader transaction throughput (1,000 TAs)

As runtime parameters, we always used 4KB database pages, 4096 4KB buffer frames in *XTC*, and cold buffers.[11] Hence, the results of all read-only benchmarks were gained with the same workload (only adjusted to the DB sizes) in the same system environment.

Figure 6 summarizes the results of our empirical experiments for read-only benchmarks running 1K transactions, which behaved in the anticipated way, at least in principle. They confirm that there is a substantial rise in transaction performance (*tps* from disk to flash configurations (configs). The results of the disk/flash configs@10MB should be excluded from this comparison, because their differences are strongly overloaded with caching effects. In this case (see Table 1), all index pages are in the XTC buffer after ~ 20 queries, whereas obviously less index locality occurs for larger DB sizes and, for configs@10GB, index locality primarily occurs at higher index levels, because not all index pages reach the buffer during the benchmark runs. On the other hand, longer search paths have to be traversed, because h_{docind} grows with the document size, and longer seeks are needed, because the storage footprint covers a larger disk area.

To illustrate the growing IO overhead, e.g. for *full* storage mapping, the 1K-transaction benchmark needed ~ 1200, ~ 2400, ~ 3430, and ~ 4550 page fetches for the 10MB, 100MB, 1GB, and 10GB sizes, respectively. Because the same set of transactions was executed and $\geq (h_{docind} + 1)$ logical page references were required in each of them, these numbers of XTC buffer faults reflect the locality achieved. As particularly visible for the IO-intensive workload at the *tps* level, scalability has suffered from the considerably growing sets of index pages and, in turn, the increasing index heights ($2 \leq h_{docind} \leq 3$ in Table 1). However, it affects to a much larger degree disk than flash configurations.

In contrast to the benchmarks with dominating IO-related activities, the CPU-intensive workloads only weakly exhibit the overall characteristics, because the IO influence with only $\sim 15\%$ of total time is almost leveled out. Of course, if hardly any IO is executed in a benchmark, flash cannot play its advantage off against disk.

[11] The effects of the kernel cache on transaction throughput could not be controlled. We observed that, when a 4KB page was requested, frequently a 64KB block was prefetched into the kernel cache.

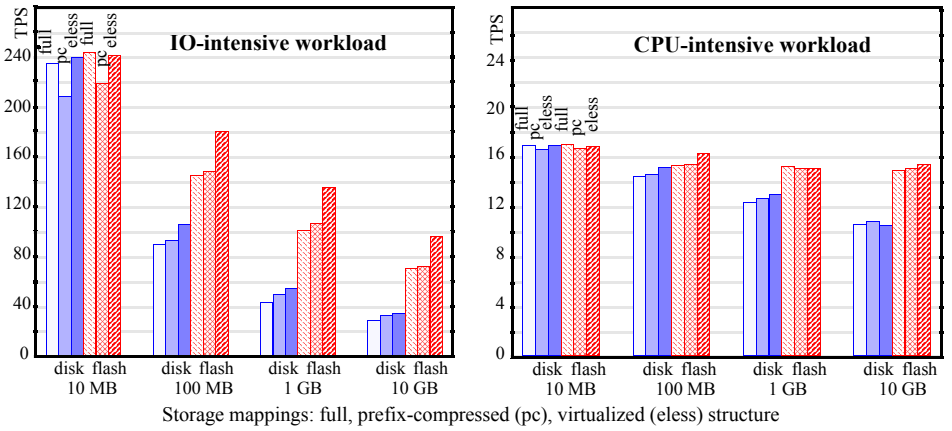


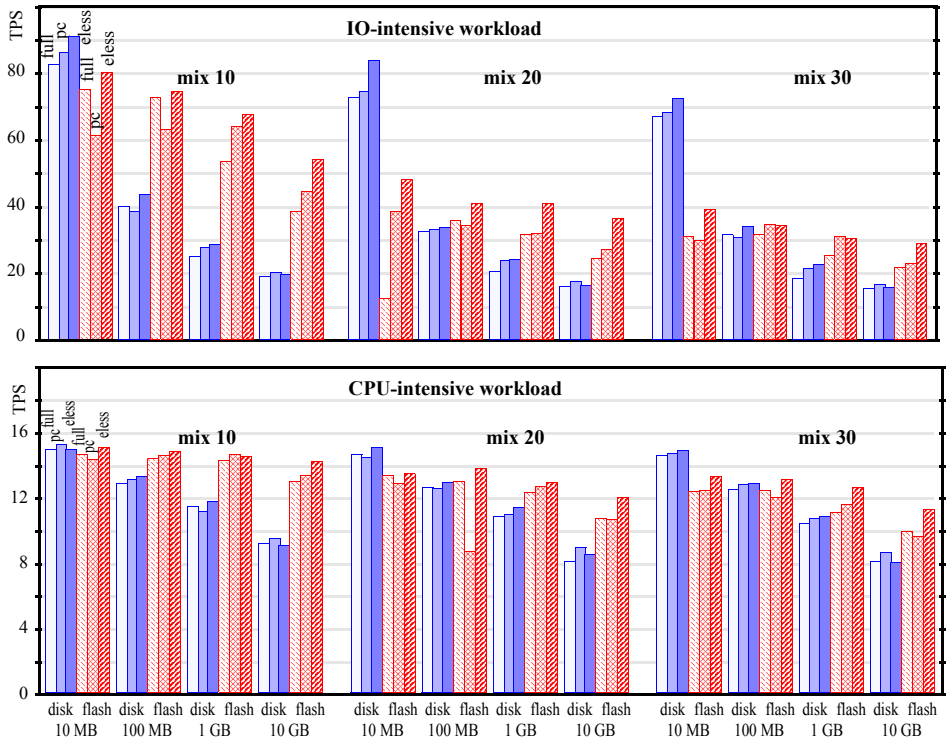
Figure 7: Reader transaction throughput (10,000 TAs)

Figure 7 summarizes the results for the benchmarks with 10K transactions, where the caching effects clearly increase on all setups and the IO-intensive *tps* rates more than double for the 10MB DB on flash as well on disk because of substantial locality on index and container pages. But also the larger DB sizes up to the configs@10GB are strongly influenced by index locality. While in the 1K-transaction, IO-intensive case the flash *tps* rates for configs@100MB and larger are $\sim 150\,\%$ – $\sim 200\,\%$ higher than for disks, they are reduced in the respective 10K cases down to $\sim 75\,\%$ – $\sim 150\,\%$—a definite example that caching may level out the flash performance superiority.

The result representation in Figure 6 and Figure 7 allows for some differential interpretations. What are the gross effects of *eless* and *pc* as compared to *full*? The *eless* mapping having the smallest storage footprint (only $\sim 70\%$ of *full*) proved its superiority in terms of *tps* throughput among the disk cases and among the flash cases. The picture is not so clear with *pc* despite a footprint of $\sim 75\%$ of *full*. This may be caused by the decompression efforts needed to reconstruct the SPLIDs in the structure part. But, in general, the built-in IO saving of a storage mapping pays off and is, in particular, visible in Figure 7 for the *eless* flash-configs@100MB and larger. While disk seeks take longer, flash access times are independent of DB sizes.

To gain some insight into application-level effects of the flash's "bad" write performance, we extended our benchmark with writer transactions. Figure 8 illustrates the results obtained for an increasing write share—again separated for the IO-intensive and CPU-intensive cases. As expected, the flash write performance continuously decreases with a growing share of writer transactions. In our experiments, the *tps* ratio steadily shrinks with an increasing writer share (from *10% write* to *30% write*). Comparing them to the read-only 1K transaction, IO-intensive case, the flash *tps* rates are clearly lower and reach $\sim 80\,\%$ – $\sim 90\,\%$ for *mix 10* and only $\sim 30\,\%$ – $\sim 35\,\%$ for *mix 30*. The performance gap to the disk closes with a growing writer share as well.

Our experiments definitely reveal the importance of the *eless* storage mapping—even more for write transactions. It enabled the highest *tps* rates in more or less all flash configurations and DB sizes.



Storage mappings: full, prefix-compressed (pc), virtualized (eless) structure

Figure 8: Transaction throughput for mixed workloads (1,000 TAs)

6.3 Energy Efficiency

So far, we have shown that at least IO-intensive (read-only) DB applications may substantially improve their transaction throughput when flash devices are used as DB store. To obtain some hints concerning the energy efficiency in our flash/disk comparison, we have started some initial measurements to figure out the overall energy consumption of our computer system (including processor, memory, cooling, and separate OS disk) when running with a flash or a disk as DB store. Table 2 summarizes the results of controlled 60-minute runs in *idle* mode and under the CPU-intensive and IO-intensive benchmarks on the 10GB DB (read-only, *pc* storage mapping).

The relative flash/disk power consumption in *idle* mode yields $P_f = \rho * P_d$ (with $\rho \sim 0.96$ in our case)[12]. To derive a formula for the same DB work, the runtimes for flash and disk can be set to $t_f = (tps_d/tps_f) * t_d$. Hence, we can compute the energy W needed for the same DB load and obtain $W_f = P_f * t_f = \rho * (tps_d/tps_f) * t_d * P_d$. If we define the flash's energy efficiency (w.r.t. disk) as $E_f = W_d/W_f$, we are able to pinpoint the essence of our

[12] Although the power continuously varies between P_{min} and P_{max}, we simplify our considerations by assuming constant consumption over time.

Table 2: Energy consumption of a computer system with flash and disk database.

Device	Workload	Throughput tps	Power (Watt) min	Power (Watt) max	Energy KWh	No. of TAs per Joule
Flash	idle	0	52.1	55.0	0.053	0
	CPU-intensive	12,530	49.5	84.6	0.071	0.176
	IO-intensive	72,240	48.8	92.8	0.085	0.850
Disk	idle	0	53.9	65.0	0.055	0
	CPU-intensive	10,470	53.7	87.9	0.063	0.166
	IO-intensive	22.810	53.7	94.7	0.062	0.368

findings summarized in Table 2. For the IO-intensive case, we yield a surprisingly large factor $E_f \leq 3.3$, whereas with $E_f > 1$ (in our case $E_f \simeq 1.25$) only moderate improvements can be gained for CPU-intensive cases.

In general, there are several possibilities to specify "efficiency", and a well-suited definition seems the (application-specific) *#TAs/Joule* ratio of a system (see Table 2). As shown in this paper, the plain IO characteristics cannot serve to choose a device for an application-level use case. Hence, the *#TAs/Joule* ratio should be addressed when thinking about efficiency. Here, flash devices have selectively shown their superiority over traditional disk storage.

7 Conclusions

The DB community will be confronted in the near future with the use of flash storage not only in embedded devices, e.g., PDAs, because it is energy efficient, small, light-weight, noiseless, and shock resistant. Expected breakthroughs in bandwidth (IO/s), energy saving, reliability, and volumetric capacity, will certainly shift this technology into high-end DB server environments. If data volumes in enterprises further grow (to the PB range), its zero-energy needs—when idle or turned off—will become a strong sales argument.

From this perspective, we started to explore the integration of flash storage into databases. We delivered a kind of "existence proof" for simple use cases where "flash beats the disk". There are definitely (read-only) scenarios where substantial performance gains and, at the same time, energy savings can be achieved. For write scenarios—as expected—, algorithmic adjustments are necessary in various DBMS components, in particular, in the areas of cache management and logging. But we could show for the flash that *performance can be kept or even improved while energy is saved*.

On the other hand, we are just at the beginning of flash-aware improvements in DBMSs. But, we believe that it is not sufficient to just develop new algorithms (e.g., for sort or join [20]) focusing on the device characteristics alone. While probably no general flash superiority is possible, there exist specific application areas which could profit from its use. Therefore, our future research will attempt to disclose these potentials and to bring flash technology to the core of database management.

References

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

[2] Bouganim, L., Jonsson, B., and Bonnet, P.: uFLIP: Understanding Flash IO Patterns. Proc. Int. Conf. on Innovative Data Systems Research (CIDR), Asilomar 2009.

[3] Butt, A. R., Gniady, C., and Hu, Y.C.: The Performance Impact of Kernel Prefetching on Buffer Cache Replacement Algorithms. Proc. SIGMETRICS Conf., 157–168 (2005).

[4] Graefe, G.: Database Servers Tailored to Improve Energy Efficiency. Proc. Int. Workshop on Software Engineering for Tailor-made Data Management, 24–28, Nantes, France (2008).

[5] Graefe, G.: The Five-minute Rule: 20 Years Later and How Flash Memory Changes the Rules. Proc. Third Int. Workshop on Data Management on New Hardware (DaMoN), Beijing, China (2007).

[6] Gray, J., and Fitzgerald, B.: FLASH Disk Opportunity for Server-Applications. http://research.microsoft.com/~Gray/papers/FlashDiskPublic.doc (Jan. 2007).

[7] Gray, J.: Tape is Dead, Disk is Tape, Flash is Disk, RAM Locality is King, powerpoint talk, Microsoft, Dec. 2006.

[8] Gray, J. and Reuter, A.: Transaction Processing: Concepts and Techniques. M. Kaufmann Publisher (1993).

[9] Härder, T. and Rahm, E.: Database Systems – Concepts and Implementation Techniques (in German). 2nd edition, Springer (2001).

[10] Härder, T., Mathis, C., and Schmidt, K. Comparison of Complete and Elementless Native Storage of XML Documents. Proc. 11th Int. Database Engineering & Applications Symposium (IDEAS), 102–113, Banff, Canada, (2007).

[11] Härder, T., Haustein, M. P., Mathis, C., and Wagner, M.: Node Labeling Schemes for Dynamic XML Documents Reconsidered. Data&Knowledge Engineering 60:1, 126–149, Elsevier (2007).

[12] Haustein, M. P., and Härder, T. An Efficient Infrastructure for Native Transactional XML Processing. Data&Knowledge Engineering 61:3, 500–523, Elsevier (2007).

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

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

[15] Leventhal, A: Flash Storage Today. ACM Queue 6:4, 24–30 (2008).

[16] MFT: Managed Flash Technology. http://www.easyco.com/mft/index.htm (2008).

[17] Miklau, G.: XML Data Repository. http://www.cs.washington.edu/research/xmldatasets.

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

[19] Nicola, M., Kogan, I., and Schiefer, B.: An XML transaction processing benchmark. Proc. ACM SIGMOD Conf., 937–948, Beijing, China (2007).

[20] Shah, M. A., Harizopoulos, S., Wiener, J. L., and Graefe, G.: Fast Scans and Joins using Flash Drives. Proc. 4th Int. Workshop on Data Management on New Hardware(DaMoN), 17–24, Vancouver, Canada (2008).

[21] SSD White Paper. http://www.supertalent.com/datasheets/SSD_WHITEPAPER.pdf

[22] XML Database Benchmark: Transaction Processing over XML (TPoX). http://tpox.sourceforge.net/ (January 2007).