Trading Memory for Performance and Energy

Yi Ou and Theo Härder

University of Kaiserslautern
ou,haerder@cs.uni-kl.de

Abstract. Managing extremely large amounts of data with high performance and low power consumption is very difficult. We look at this urgent problem from an architectural perspective and present our prototype design and implementation of a three-layer database storage system, which uses flash-based devices as an intermediate caching layer. The flash-based layer significantly improves the I/O efficiency of the storage system. Therefore, we can reduce the use of energy-inefficient RAM-based memory without compromising the overall system performance. The efficiency of the three-layer storage system is demonstrated by our practical experiments using traces from both standard benchmarks and a real-life application.

1 Introduction

The worldwide data volume is doubling every two years. According to the estimation of IDC, currently 45 GB of data in average exists for each person in the world: that is 281 Billion GB (281 Exabytes) in total. At the same time, IT enterprises are still hungry for data [1]. To cope with the pace of data explosion, the number of installed servers and storage systems is rapidly growing, resulting in a huge amount of energy consumption. One of the greatest challenges for the information management community is to manage extremely large amounts of data in an (both performance and energy) efficient way.

Flash memory is a kind of non-volatile storage media, popularly used in memory cards and USB flash drives. In the desktop PC and server storage markets, solid-state disks based on flash memory (flash SSDs) are also gaining attention, due to their increasing storage capacity and decreasing price. In contrast to traditional hard disk drives (magnetic HDDs), flash SSDs have no mechanical parts and, therefore, allow much faster random accesses. Because flash memory is non-volatile, its active power is much lower (compared on a Watt/GB basis) than that of DRAM, for which a large portion of the active power is consumed to maintain the state of the chip.

Currently, most of the database storage systems follow a classical *two-layer architecture* (2LA) [2], with a RAM-based buffer layer accelerating page requests to and from the persistence layer based on hard disk drives. With an increasing amount of data accommodated at the persistence layer, the capacity of the expensive and energy-inefficient RAM-based buffer typically becomes the performance bottleneck.

Table 1: Price and performance of storage devices

Device	Model No.	EUR/GB	Read (ms)	Write (ms)
RAM1	Kingston KVR667D2D8P5/2G	19.00	$\sim$ 10 ns	$\sim$ 10 ns
RAM2	Kingston KHX1600C9D3B1K2/4GX	19.11	$\sim$ 10 ns	$\sim$ 10 ns
RAM3	Kingston KVR1333D3D4R9S/4G	24.70	$\sim$ 10 ns	$\sim$ 10 ns
SSD1	Intel SSDSA2MH160G1GN	2.40	0.029	0.303
SSD2	Intel SSDSA1MH160G2GN	2.44	0.029	0.116
SSD3	Crucial CTFDDAC256MAG-1G1	2.01	0.017	0.022
HDD1	WD WD800AAJS 7200 RPM	0.38	15.000	15.000
HDD2	WD WD1500HLFS 10000 RPM	0.77	4.500	4.500
HDD3	Fujitsu MBA3147RC 15000 RPM	0.76	2.000	2.000

In terms of performance and price and their long-term trend, flash SSDs fit perfectly into the gap between DRAM and magnetic HDDs. Table 1 lists, for each storage media type, the prices and performance figures[1] of three devices (from low-end to high-end). These figures strongly suggest a *three-layer architecture* (3LA), where flash is used as an intermediate caching layer, while conventional and inexpensive HDDs are employed at the bottom layer to accommodate our ever-increasing demand on storage capacity. With such a memory hierarchy, the capacity of the RAM-based layer could be kept relatively small, because a larger amount of pages can be cached on the flash media, which is still much faster than HDDs. To justify the move from 2LA to 3LA, a few questions need to be answered:

Q1 Will the cost of adding the intermediate layer be justified by performance improvements?
Q2 Can we achieve the goal of improving performance while saving energy at the same time?

The major contribution of this paper is giving answers to the questions Q1 and Q2, which are ignored so far in related works. In addition, we contribute in the following aspects:

- We advocate 3LA, the three-layer database storage architecture with flash as the intermediate layer.
- We define the basic interfaces for the three layers and present the prototype of such a storage system.
- Using buffer traces of standard benchmarks and a real-life workload, we accomplish an extensive empirical study, comparing the performance and energy consumption of 2LA and 3LA.

The remainder of this paper is organized as follows: Section 2 discusses related works. Section 3 presents our design of 3LA and related algorithms. Sec-

[1] We used the sales prices of Internet stores as of November 2010. Performance figures are derived from the device data sheets, for randomly accessing pages of 4 KB.

tion 4 reports our empirical study. The concluding remarks and future works are presented in Section 5.

2 Related Work

Multi-level caching has been intensively studied in the past. Zhou et al. [3] characterized second-level buffer access patterns and proposed a set of algorithms for managing the second-level buffer. Those algorithms are not flash-specific, therefore, their major performance metric is the hit ratio. One of them is implemented in our prototype system and included in our experiments.

Koltsidas and Viglas [4] identified three page-flow schemes in a three-level caching hierarchy and proposed flash-specific cost models for those schemes. While addressing both theoretical problems and important implementation issues, their focus is the validation of the cost models and the comparison among those schemes. Energy efficiency and a comparison between 2LA and 3LA are not covered in their work.

Narayanan et al. [5] addressed both complete replacement of disks by SSDs, as well as use of SSDs as an intermediate tier between disks and DRAM. They compare these architectural variants with 2LA using an offline tool, which, given a block-level trace of a workload, suggests the least-cost storage configuration that supports the workload's requirements. They found that replacing disks by SSDs is not a cost-effective option for any of their workloads, due to the higher dollar-per-GB cost of flash SSDs.

Although our goal partially overlaps with that of [5], there are several aspects that distinguish our work fundamentally from theirs: 1. Their traces represent the workload to the disk layer (block level), while our traces represent the workload to the buffer manager (buffer traces). 2. Our observations are quite different from theirs. For example, they found that fewer than 10% of their workloads can benefit from an intermediate layer based on flash, while in our experiments, 3LA is superior to 2LA in most configurations. 3. Our observations are expected to be more accurate, because in their experiments, traces were not executed, but just analyzed by the tool, while our traces are actually run in the real systems.

3 The 3LA Storage System

We consider three layers of software in our storage system, as shown in Figure 1. The RAM layer L_r manages the buffer pool with $|L_r|$ pages in main memory, the flash layer L_f manages the flash-based buffer pool with $|L_f|$ pages, and the disk layer L_d manages the accesses to the magnetic disk or a pool of (possibly inexpensive and redundant) magnetic disks with a total capacity of $|L_d|$ pages.

Considering the relative price and performance ratios of the three types of storage media, e. g., those listed in Table 1, we assume that:

$$|L_r| \leq |L_f| \leq |L_d| \tag{1}$$

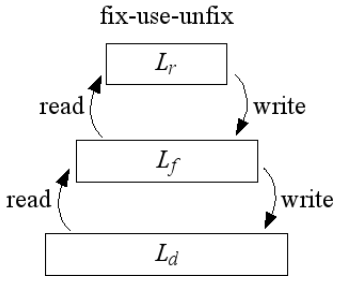


Fig. 1: Inter-layer interfaces in the three-layer architecture

Due to these capacity constraints and performance ratios, the hottest pages should be kept in L_r, and L_f should try to keep the hot pages that can not be kept in L_r. As a consequence, replacement policies are required in L_r and L_f.

L_r supports a typical buffer pool interface, e. g., that of the classical fix-use-unfix protocol [6]. Both L_f and L_d provide the interface of reading or writing a page, identified by its logical page number. Each layer only uses the interface provided by the layer directly below it, i. e., there is no cross-layer dependency. In particular, in 3LA, L_r never accesses L_d directly.

However, because L_f and L_d basically have the same interface, L_f can be implemented as an optional layer. When L_f is not present, L_r directly accesses L_d. In that case, 3LA degenerates to 2LA. Such a degeneration is practically used for our experiments in Section 4.

For both architectures, we assume that L_r follows two basic principles: *demand paging* and *write back*. Consequently, we have the following two *invariants*, which are independent of the algorithm and implementation of L_f and valid for both 3LA and 2LA:

I1 L_r calls the read(p) function on the layer directly below it, iff page p is not present in L_r and there is a page request for p to be served by L_r (page fault in L_r).

I2 L_r calls the write(p) function on the layer directly below it, if page p is to be evicted from L_r and p is dirty (modified at least once after entering L_r).

L_r and L_d are basically the same as in the classical two-layer disk-based storage system. For this reason, we only present the replacement algorithms for the management of L_f in the following: the *Local* (LOC) algorithm and the *Global* (GLB) algorithm.

In both algorithms, a list of cache positions L with $|L| = |L_f|$ is maintained in an LRU fashion. A *cache position* identifies a page slot in the flash-based cache and contains a clean/dirty bit. Furthermore, a directory H is maintained, mapping currently cached pages to their corresponding cache positions.

3.1 The LOC Algorithm

In the LOC algorithm, L_f is managed *locally* in an LRU fashion, without requiring extra knowledge from L_r. The procedure of reading a page from L_f is shown in Algorithm 1. One difference to an main-memory LRU cache is that flushing a page involves first reading the page from flash and then writing it to the storage. Writing a page p to L_f involves finding its cache position c via H and storing p at c. If p is not found in L_f, it will be written to L_d immediately.

Because $|L_r| \leq |L_f|$ and LOC only has local knowledge, it is possible that some or even all pages in L_r are doubly cached in L_f. However, pages in L_r are not necessarily all in L_f, due to different page reference behaviors at different layers. Note, references to L_f are consequences of buffer faults in L_r.

Algorithm 1: LOC read page from L_f

```
   input  : read request for page p, storage layer L_d
   output : update L and H; return p with content loaded
 1 cache position c ← lookup p in H ;
 2 if c ∈ L_f then
 3 |   read p from cache position c ;
 4 |   move c to MRU position of L ;
 5 else
 6 |   victim cache position v ← LRU position of L ;
 7 |   page q ← the page stored at v ;
 8 |   if v is dirty then
 9 |   |   read q from cache position v and flush q to L_d ;
10 |   read p from L_d and store p at v ;
11 |   move v to MRU position of L ;
12 |   update H by replacing entry (q, v) with entry (p, v) ;
13 return p;
```

3.2 The GLB Algorithm

The GLB algorithm is first introduced in [3]. We examine it here in a flash context. The GLB algorithm follows the exclusive scheme [4], i. e., no page is ever cached in L_r and L_f at the same time. For better comprehension, we assume the replacement policy in L_r is also LRU, without loss of generality. Based on this assumption, we can think of a *global* logical LRU list L_g, consisting of the LRU list of L_r at its MRU end, and the LRU list of L_f at its LRU end.

Reading a page p from L_f is requested upon a page fault in L_r (see I1). In case of a cache hit in L_f, p is moved from L_f to L_r (H and L are updated accordingly). In case of a cache miss, p is read directly from L_d to L_r, avoiding doubled caching in L_f. In both cases, a page q is evicted from L_r to L_f. After being read, p becomes the MRU page in L_r (also in L_g).

To "maintain" the logical list L_g, page q currently evicted from L_r should become the LRU page in L_f. Therefore, we have to extend the interface of L_f (as described in Figure 1) by a new function *evict* called by L_r for passing evicted clean pages to L_f. Note, a write request is called on L_f, only when the evicted page is dirty (see I2). The procedure of processing a write or evict request for page q is the same: flush the page stored at the LRU position v of L if the page is dirty, move v to the MRU position, store q at v, mark v dirty if q is dirty, and update H.

3.3 Discussion

Given the same workload, the global cache hit count (total number of buffer hits in L_r and L_f) of GLB is expected to be higher than that of LOC, because the effective cache size of the latter is smaller, due to doubled caching in L_r. However, in GLB, the number of flash writes equals the number of L_r page evictions. This is OK for a RAM-based second-level buffer, but it is an issue for flash media both in terms of performance (see Section 4) and lifespan [7].

For both algorithms in our current implementation, the dirty pages in L_f (whose cache positions are marked dirty) are flushed to L_d when the system is shutdown, for the sake of consistency. A simple improvement leveraging the non-volatility of flash can be made here: we can just materialize the content of H at shutdown and rebuild H at startup[2], without flushing the "dirty" pages in L_f. This technique not only speeds up the shutdown procedure, but also shortens the warm-up phase of the system, because the hot portion of the pages are likely already in L_f, ready for immediate access. For the LOC algorithm, pages in L_f are up-to-date at restart, iff the dirty pages of L_r are flushed before the shutdown of L_f starts. For the GLB algorithm, page sets L_f and L_r are disjunct, therefore, pages in L_f are automatically up-to-date at restart.

4 Experiment

To answer the questions Q1 and Q2, we did an extensive empirical study based on a fair comparison between 2LA and 3LA, using buffer traces recorded under various workloads. We first present our simulation-based study using TPC-E, TPC-C, and TPC-H traces, before we discuss the experiment ran on real devices using the trace from a real-life application. Our study on energy consumption is based on the following assumption:

A1 The acquisition cost and power consumption of storage media are linear to their capacity in use.

Assumption A1 might not be valid at fine granularity, however, it is reasonable, when observed at a coarser granularity. For example, if the power of a 2-GB DRAM module is 10 W, according to A1, 0.2 GB of DRAM would consume 1

[2] The byte size of H is much smaller compared to that of L_r and L_f.

W, which is not valid, because, as long as the module is working, it consumes 10 W, no matter the remaining 1.8 GB are in use or not. But we can safely say that $2n$ GB of DRAM based on the same model consume $10n$ W.

All experiments were done using our prototype implementation of the 3LA storage system, which can also be easily configured to function as a 2LA system, as described in Section 3. For both architectures, our test program only communicates with L_r by sending the logical page requests delivered by the traces to its buffer manager, which manages the L_r buffer pool using the replacement policy LRU. All experiments start with *cold* L_r and L_f buffers. The time used to flush the dirty pages at shutdown is included in the measurements.

In our experiments, we scaled a parameter b (in number of pages) logarithmically. For 2LA, b is the size of the buffer layer, i. e., $|L_r| = b$, while for 3LA, we set $|L_r|$ and $|L_f|$ as follows:

$$|L_f| = n \times b \tag{2}$$

and

$$|L_r| = max(1, \lfloor b - |L_f| \times (C_f/C_r + S_d/S_p) \rfloor) \tag{3}$$

where C_f/C_r is the dollar-per-GB cost ratio of flash to RAM, S_d is the byte size of a directory entry of H, and S_p the page size in bytes. The term $|L_f| \times C_f/C_r$ gives the number of RAM pages that should be reduced to achieve a cost-neutral investment for $|L_f|$ pages of flash memory. The term $|L_f| \times S_d/S_p$ is the number of RAM pages consumed by the directory H for $|L_f|$ pages of flash. We call Formula 2 and 3 the *equi-cost constraints*, because it enforces a fair basis for the comparison among the 2LA and 3LA configurations, i. e., having the same acquisition cost.

The parameter n is used to examine the behavior of 3LA when the size of L_f is scaled. Because the value of $|L_r|$ can not be negative, we have $b - |L_f| \times (C_f/C_r + S_d/S_p) > 0$, which resolves to $n < 1/(C_f/C_r + S_d/S_p)$. Together with the constraint in Formula 1, we have the practical range of n:

$$1 \leq n < (C_f/C_r + S_d/S_p)^{-1} \tag{4}$$

If we ignore S_d/S_p, which is relatively small, then we obtain $1 \leq n < C_r/C_f$. In our experiments, the page size S_p is 8192 bytes and the directory entry size S_d is 4 bytes. We chose the cost ratio $C_f/C_r = 0.10$, which is very close to the real price ratios according to Table 1. According to Formula 4, the practical range of n is approximately $[1, 10)$. Note n does not have to be an integer. For a given b, the value of n actually controls how much RAM is traded for flash, observing the equi-cost constraints.

4.1 Simulations

For the simulation-based experiments, the *Virtual Execution Time* (T_v) is used as the major performance metrics, defined as:

$$T_v = T_f + T_d \tag{5}$$

Here, T_f and T_d are the simulated device access times elapsed in L_f and in L_d respectively. T_f is defined as:

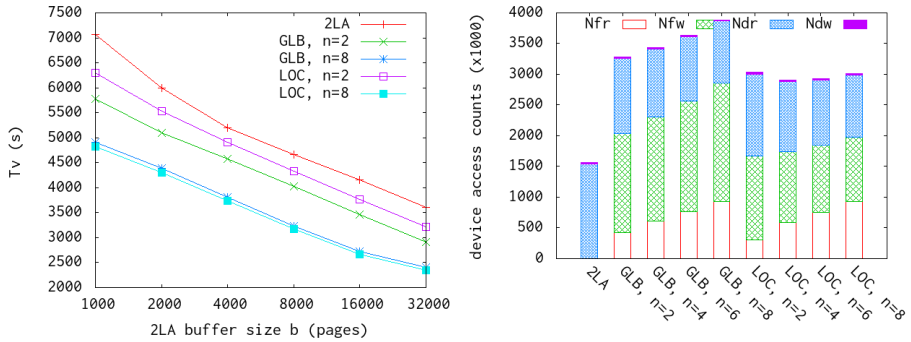
$$T_f = T_{fr} + T_{fw} = N_{fr} \times C_{fr} + N_{fw} \times C_{fw} \tag{6}$$

T_{fr} and T_{fw} are the accumulated times reading from and writing to the flash media, N_{fr} is the number of flash reads, C_{fr} the average cost of a flash read, N_{fw} the number flash writes, and C_{fw} the average cost of a flash write. The flash reads and flash writes here refer to the physical reads from and writes to the flash device. They are not to be confused with the read and write requests sent to the L_f software. Similarly, T_d is defined as:

$$T_d = T_{dr} + T_{dw} = N_{dr} \times C_{dr} + N_{dw} \times C_{dw} \tag{7}$$

The definition of T_v only considers the costs of accessing the storage media and ignores the CPU cost, because all the algorithms involved have a constant complexity. The inter-layer communication costs are ignored as well, because, the dominating cost in the system is the cost of page accessing, not page transferring. In our simulation, we used the average read and write costs close or equal to those of the middle-class devices in Table 1, i. e., $C_{fr} = 0.030$, $C_{fw} = 0.120$, $C_{dr} = 4.5$, and $C_{dr} = 4.5$ (ms).

Figure 2a illustrates the T_v of running a TPC-E trace[3] using 2LA and 3LA. All 3LA configurations tested significantly outperform the 2LA configuration. For better clarity of the chart, we only show the curves for $n = 2$ and $n = 8$. For the $n = 8$ configuration, LOC reduced the virtual execution time by 32% to 35% (for $b = 1000$ to $b = 32000$), compared with 2LA.



(a) T_v (in seconds) **(b)** Device accesses for $b = 1000$

Fig. 2: TPC-E trace performance

[3] Provided by a leading IT enterprise.

The behavior of 3LA is better explained by Figure 2b, where the numbers of device accesses[4] are compared for $b = 1000$. For 2LA, there is no flash device access, while a significant amount of flash device accesses is required for 3LA (Figure 2b). For both GLB and LOC, with n scaled from 2 to 8 (thus an increasing $|L_f|$ and decreasing $|L_r|$), the number of flash reads climbs up, indicating a growing number of hits in L_f, and, consequently, the number of disk reads goes down. The latter is equal to the number of *global cache misses* (i. e., a page is neither in L_r nor in L_f). Because of the speed difference of flash to disk, the flash accesses introduced at L_f are paid off in terms of overall performance (Figure 2a).

As shown in Figure 2b, the number of flash writes performed by GLB increases with an increasing $|L_f|$ and a decreasing $|L_r|$, because it depends on the latter, as discussed in Section 3.3. In contrast, the increasing $|L_f|$ reduces the number of flash writes performed by LOC. This is because it reduces the number of L_f cache misses and each cache miss requires a flash write (line 10 of Algorithm 1).

Table 2: Energy consumption of the TPC-E trace for $b = 1000$

Alg.	n	$\|L_f\|$	$\|L_r\|$	P_f (mW)	P_r (mW)	$P_f + P_r$ (mW)	T_v (s)	E (J)
2LA		0	1000	0.000	4.121	4.121	7059	29.09
GLB	2	2000	799.02	0.014	3.292	3.307	5776	19.10
GLB	4	4000	598.05	0.029	2.464	2.493	5304	13.22
GLB	6	6000	397.07	0.043	1.636	1.679	5061	8.50
GLB	8	8000	196.09	0.057	0.808	0.865	4905	4.24
LOC	2	2000	799.02	0.014	3.292	3.307	6305	20.85
LOC	4	4000	598.05	0.029	2.464	2.493	5372	13.39
LOC	6	6000	397.07	0.043	1.636	1.679	5024	8.44
LOC	8	8000	196.09	0.057	0.808	0.865	4818	4.17

Table 2 compares the energy efficiency of 2LA and 3LA for $b = 1000$. The $|L_f|$ and $|L_r|$ values in the 3rd and 4th column are calculated according to Formula 2 and 3. Having these values, we can compute the power value of L_r, based on assumption A1, as follows:

$$P_r = |L_r| \times S_p \times P_r^u \tag{8}$$

where P_r^u is the unit power of RAM, having the value 0.503×10^{-9} (W/B) here, derived from the data sheet of RAM2 in Table 1. The power value of L_f, denoted as P_f, is calculated in a similar way, with $P_f^u = 0.873 \times 10^{-12}$ (W/B), derived from the data sheet of SSD2. Having $P_r + P_f$ and the virtual execution times (T_v), we can then calculate the energy consumption values in the last column. Note that the buffer layer of 2LA consumed much more energy than

[4] In the simulation, no real device access occurs.

those of 3LA (by a factor of six for $n = 8$). Disk-layer values are not included in the table, because they are of the same size in both architectures.

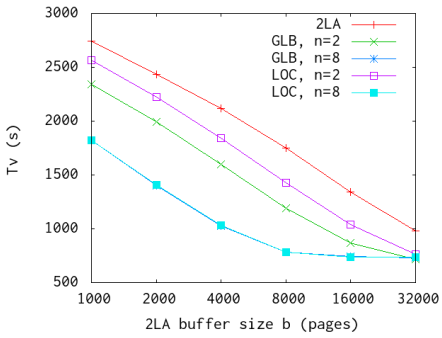


Fig. 3: TPC-C trace performance

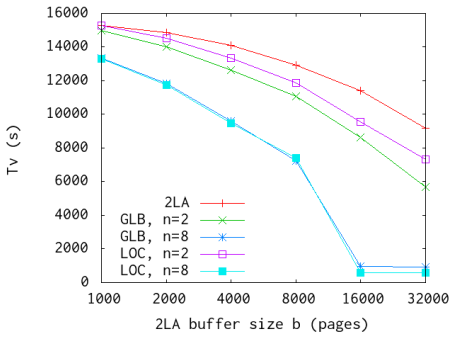


Fig. 4: TPC-H trace performance

The results of running the buffer traces of a TPC-C and a TPC-H workload are shown in Figure 3 and Figure 4. In general, these results confirm our observation on the performance advantage of 3LA. For both traces, with b beyond 16000 pages and $n = 8$, the flash cache of 3LA is large enough to accommodate all pages of the working sets, which are much smaller than that of the TPC-E trace, therefore, no performance improvement can be observed when b is increased to 32000 pages. The TPC-H trace is highly read intensive, with only 256 page updates out of 6.5 million page requests. That is the reason why the performance of 3LA improves much faster with the growing buffer sizes under the TPC-H workload (Figure 4), compared to the TPC-E and TPC-C cases.

4.2 Running a Real-Life Trace on Real Devices

Complementary to our simulation-based study, we also experimented with a trace from a real-life application on real devices. Our test machine is equipped with an AMD Athlon Dual Core Processor, 1 GB of main memory, and is running Linux (kernel version 2.6.24). HDD2 from Table 1 is used as the storage device in L_d, and SSD2 is used as the flash device in L_f. Both devices are accessed as raw devices, i. e., no file system or OS caching is involved, and our storage system has the control over the access to the devices.

The trace used here is a one-hour page reference string of an OLTP production system of a bank. This trace is well-studied and has been used in [8,9,10,11,12]. It contains 607,390 references to 8-KB pages in a database having a size of 22 GB, addressing 51880 distinct page numbers. About 23% of the requests update the page referenced.

The measured execution times (wall-clock times) are shown in Figure 5. The curves have a shape very similar to that of Figure 3, confirming the accuracy

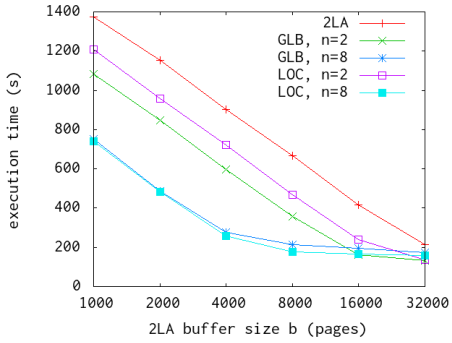
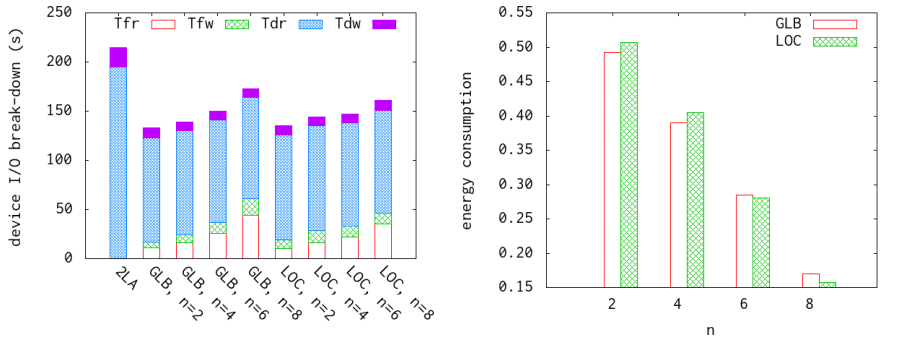


Fig. 5: Execution time (seconds) of the bank trace

of our simulation. An interesting observation can be made here: for $b = 32000$, the execution time in 3LA increases with n, instead of decreasing with it as in most cases tested. In our case here, the 51880 distinct pages addressed by the trace can be completely accommodated by L_r and L_f, for $n = 2$. Therefore, in such a situation, trading RAM for more flash does not further avoid any access to the disk layer, but reduces the number of buffer hits in L_r and introduces higher numbers of flash accesses, as indicated by Figure 6a, where a break-down of device I/O is presented, with measured values of T_{fr}, T_{fw}, T_{dr}, and T_{dw}. Nevertheless, the energy consumption decreases with an increasing n, as shown in Figure 6b, which illustrates the energy consumption figures, obtained similarly to those of Table 2, in a relative fashion.

A question arises here: how much RAM should be traded for flash? Or, in our context, what is the break-even point for n? Analytically determining the optimal value for n is a very difficult problem. However, based on our empirical research, we know that for workloads having a small working set that can be kept in the RAM layer, there is no performance benefit of trading RAM for flash, while for workloads with larger working sets that can not fit into the main memory, a larger n generally improves performance as well as energy efficiency. Of course, when n closely approaches C_r/C_f, $|L_r|$ becomes 1 (Formula 3), i. e., the RAM layer has only one page. Such extreme cases should obviously be avoided in system design. Together with Formula 4, our observations can be used as *rules of thumb* in practical applications.

Based on our experiments discussed so far, we can summarize the characteristics of GLB and LOC as follows. For small $|L_f|$, i. e., $|L_f| \sim |L_r|$, GLB achieves higher hit ratios, while for large $|L_f|$, i. e., $|L_f| \gg |L_r|$, LOC is generally better, because GLB's advantage in hit ratios becomes insignificant and it is eaten up by its higher number of flash writes, which is much more expensive than flash reads. A configuration with $|L_f| \gg |L_r|$ is closer to our goal of managing extremely large amounts of data with high performance and low power consumption.



(a) Device I/O break-down **(b)** Energy consumption relative to 2LA

Fig. 6: Statistics running the bank trace for $b = 32000$

5 Conclusion and Future Work

In this paper, we looked at the problem of using flash as a caching layer between RAM and HDDs from a new perspective: the amount of expensive and energy-inefficient RAM can be reduced due to the support of flash. Our empirical study considered the most important aspects of TCO (Total Cost of Ownership) of a storage system: the acquisition cost and the operating cost (power cost). Our study gives positive answers to the questions Q1 and Q2 and reveals that we can build a 3LA system which is much faster and much more energy efficient than a 2LA system built with the same acquisition cost, meeting the goals of performance and energy efficiency, which are often considered conflicting, at the same time.

In practice, with improved storage system performance, the number of disks, which is sometimes higher than necessary in favor of disk I/O throughput, can generally be reduced, resulting in further operational cost savings due to reduced floor space and cooling requirements.

The performance advantage of 3LA comes from the superior performance/price ratio of flash devices compared with HDDs[5]. This ratio will steadily increase in the next years, while the performance/price ratio of HDDs will remain relatively stable. As a consequence, the performance advantage of 3LA will be even more significant in the future.

LOC and GLB served as the baseline algorithms. No flash-specific optimizations are yet integrated. Techniques such as using different page size at different layers as those discussed in [4] could further improve the performance of 3LA. It could also be interesting to examine hybrid configuration of algorithms, e. g., frequency-based algorithm at one layer and recency-based algorithm at the other

[5] Similar observations are made in our experiments using the values of HDD3, the high-end HDD in Table 1.

layer. As future work, we will also look into such optimizations. However, future improvements expected for performance and energy-efficiency of 3LA do not conflict with our observations made in this paper.

One of the major differences of a flash-based cache to a RAM-based cache is non-volatility. We have discussed a technique leveraging this property to shorten the warm-up phase of the system in Section 3.3, which is not the main focus of this paper and will be empirically evaluated in the future. The non-volatility of the flash layer should be further exploited to speed up processing of transactions, for which durability is required.

6 Acknowledgement

We are grateful to IBM (Deutschland and USA) for providing the TPC-E trace and to anonymous referees for valuable comments. This research is partly supported by the German Research Foundation and the Carl Zeiss Foundation.

References

1. Spiegel. Google-chef will noch mehr daten. `http://www.spiegel.de/netzwelt/netzpolitik/0,1518,716204,00.html`, 2010.
2. T. Härder. DBMS architecture - the layer model and its evolution. *Datenbank-Spektrum*, 13:45–57, 2005.
3. Y. Zhou, Z. Chen, et al. Second-level buffer cache management. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):505–519, 2004.
4. I. Koltsidas and S. D. Viglas. The case for flash-aware multi-level caching. Technical Report, 2009.
5. D. Narayanan, E. Thereska, et al. Migrating server storage to SSDs: analysis of tradeoffs. In *EuroSys*, pages 145–158. ACM, 2009.
6. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1993.
7. E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys (CSUR)*, 37(2):138–163, 2005.
8. E. J. O'Neil, P. E. O'Neil, et al. The LRU-K page replacement algorithm for database disk buffering. In *SIGMOD*, pages 297–306, 1993.
9. T. Johnson, D. Shasha, et al. 2Q: a low overhead high performance buffer management replacement algorithm. In *VLDB*, pages 439–450, 1994.
10. N. Megiddo and D. S. Modha. ARC: A self-tuning, low overhead replacement cache. In *FAST*. USENIX, 2003.
11. Z. Li, P. Jin, et al. CCF-LRU: A new buffer replacement algorithm for flash memory. *Trans. on Cons. Electr.*, 55:1351–1359, 2009.
12. Y. Ou and T. Härder. Clean first or dirty first? a cost-aware self-adaptive buffer replacement policy. In *IDEAS*, Montreal, QC, Canada, 2010.