Evaluation of Pointer Swizzling for Database Buffer Management

Max Gilbert
University of Kaiserslautern
Germany
m_gilbert13@cs.uni-kl.de

Caetano Sauer
University of Kaiserslautern
Germany
csauer@cs.uni-kl.de

Theo Härder
University of Kaiserslautern
Germany
haerder@cs.uni-kl.de

ABSTRACT

To achieve high performance, the lower system layers of a DBMS should enable very efficient access to the database. With the exception of MMDBS, all other architectures need to cope with two levels of data access, where the database is stored on secondary storage and (part of) the current working set is kept in main memory and managed in the DBMS buffer. Even if ever larger main memories are available today, the performance behavior of most DBMS still relies on an effective buffer management, where only a fraction of an OLTP workload is present in main memory.

Therefore, DB buffer management is usually optimized for scenarios where only a small subset of the database is held in main memory and still many references to database pages require pages to be accessed from secondary storage. Those page misses take much longer (typically orders of magnitude) than page hits, when the referenced page is found in the DB buffer. Optimization efforts concerning page hits do not result in substantial performance gain when a significant amount of references results in page misses. On the other hand, when (almost) only page hits occur, even a tiny performance optimization of those page accesses can bring substantial performance improvement—even if that results in a slightly increased overhead on page misses. In this paper, we evaluate the use of pointer swizzling by a DB buffer manager as a measure to optimize page hits in a mostly memory-resident OLTP workload.

Keywords

Database Buffer Management, Pointer Swizzling

1. INTRODUCTION

Durability as an essential demand of the ACID paradigm requires a persistent state of the DB, usually achieved by use of “stable storage” (HDDs or SSDs). A DBMS, in turn, needs fast and efficient access to DB data while processing DB operations. For this reason, buffer management provides in-memory copies of DB pages to the higher DBMS layers.

The access time gap between RAM and SSDs or even HDDs makes page misses in a buffer pool tremendously more expensive than page hits. Thus, an obvious optimization goal is to augment the hit rates, which can be achieved—besides improved page replacement algorithms, often tailored to the workload—by use of ever larger DB buffers. These days, an OLTP working set often completely fits into the DB buffer pool due to grown main-memory sizes; therefore, I/O latency is not a bottleneck anymore. But, because of the substantially increased average speed of page references, the processing overhead of page hits may now become relevant for the overall DBMS performance. It might even become the major DBMS bottleneck, as the results of S. Harizopoulos et al. [5] suggest.

The operations executed by the buffer manager during a page reference are *fix* and *unfix*. In case of a page hit, i.e., if the page is found in the buffer pool, the buffer manager needs to locate the page stored in one of its buffer frames; then, it acquires the related frame latch for the calling thread and modifies the statistics of the page replacement module (e.g., increase the reference count or update the LRU list). During unfix, the only operation required is to release the latch, which has insignificant cost. The update of page replacement statistics is usually also cheap with a suitable strategy like CLOCK, which requires writing to a local variable instead of modifying a global data structure, as in LRU. Therefore, the only two significant operations during a page hit are latching and mapping page IDs to frames. This work focuses on the latter aspect, i.e., on reducing the cost of page ID lookups for page hits.

The technique explored in this work is *pointer swizzling* in the context of buffer management. Section 2 outlines the process of page ID lookups in detail, while Section 3 describes how this process is optimized with pointer swizzling. Section 4 presents a short performance evaluation of pointer swizzling compared to traditional approaches. Finally, Section 5 concludes this paper.

2. USAGE OF A HASH TABLE

2.1 Fix a Page

When a hash table is used to locate database pages in the buffer pool (and latching for concurrency control), a page fix basically works as sketched by Algorithm 1. First, the hash table is looked up for the given page ID on line 2. Such a hash table is illustrated in Figure 1. The hash table returns the index of the buffer frame, where the requested page can be found, or *NIL*, if the page is not in the buffer pool.

29th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 30.05.2017 - 02.06.2017, Blankenburg/Harz, Germany.
Copyright is held by the author/owner(s)..

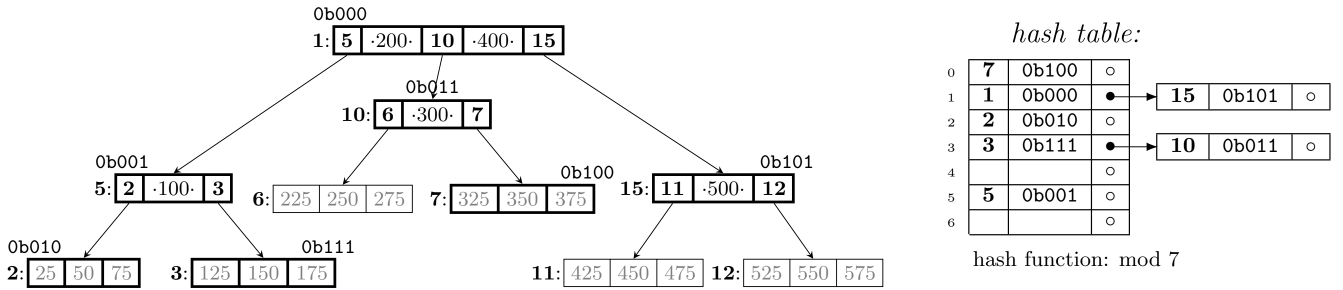

Figure 1: Example of a B+tree-like data structure partially buffered in main memory where the buffer manager uses a hash table to locate the buffered pages. The notation is defined in Figure 2.

```
 1: function FIXPAGE(pageID, latchType)
 2:   frameID ← HASHTABLE.LOOKUP(pageID)
 3:   if frameID = NIL then                              ▷ Miss
 4:     if FULL then
 5:       frameID ← CHOOSEVICTIM
 6:       ACQUIRELATCH(frameID, EX)
 7:       victimPageID ← GETPAGEID(frameID)
 8:       HASHTABLE.REMOVE(victimPage)
 9:     else
10:       frameID ← CHOOSEFREEFRAME
11:       ACQUIRELATCH(frameID, EX)
12:     end if
13:     page ← STORAGEMANAGER.RETRIEVE(pageID)
14:     INSERTINTOFRAME(frameID, page)
15:     HASHTABLE.ADD(pageID, frameID)
16:   end if
17:   ACQUIRELATCH(frameID, latchType)
18:   REFERENCESTATISTICS(frameID, pageID)
19:   return GET(frameID)
20: end function
```

Algorithm 1: Page fix without pointer swizzling

If a page miss happened, the page needs to be fetched from secondary storage, which requires a free buffer frame. If there is none (line 4), then a page needs to be evicted. On line 5, a page gets selected for eviction using some page replacement algorithm. To prevent concurrent fixes of the page to be evicted, the corresponding frame needs to be exclusively latched (line 6). Afterwards, it can be removed from the hash table using its page ID retrieved on line 7. If there is a free buffer frame, its index gets retrieved on line 10. To allow the buffering of a new page inside that frame, it requires again an exclusive latch (line 11).

When the index of such a free frame is known, the requested page can be retrieved from secondary storage. The required I/O operations are handled by the storage manager on line 13. The actual page is written to the free buffer frame on line 14 and its page ID is inserted into the hash table on line 15 to enable the location of the page during subsequent fixes of that page.

When a requested page is already in the buffer pool, the related latch is acquired[1] on line 17. Most page replacement strategies collect statistics on page fixes; thus, the fix operation is recorded on line 18. Finally, a reference to the requested page is returned on line 19.

[1]If it was a page miss, the held exclusive latch might get downgraded.

2.2 B+Tree Traversal

Let us discuss the search in a B+tree index using an example where the initial state of the buffer is shown in Figure 1. The left part shows a B+tree index structure partially cached in the DB buffer. The right part visualizes the corresponding hash table, mapping each of the buffered database pages to its corresponding buffer frame.

When a record with key 250 is requested, the search in the B+tree starts at the root page. The page ID of the root—**1**—is given from the system catalogs. Therefore, the first call to the buffer manager will be FIXPAGE(**1**, SH). The call of LOOKUP(**1**) calculates $1 \equiv (1 \bmod 7)$ and the hash table's entry for a hash value of 1 is the mapping from page ID **1** to the frame index `0b000`. Because of the encountered page hit, the only major step left is to acquire a shared latch using ACQUIRELATCH(`0b000`, SH). If no other thread holds an exclusive latch on this frame, this call will return nearly instantaneously.

The search is continued within page **1** and identifies the next page to be examined having page ID **10**. Therefore, the buffer manager will call be FIXPAGE(**10**, SH) which, in turn, will execute LOOKUP(**10**) on the hash table, yielding $10 \equiv (3 \bmod 7)$; thus, the frame index corresponding to page ID **10** is in the hash table at index 3. However, the found entry is for page ID **3**. Therefore, the related overflow chain has to be checked, revealing that the searched page can be accessed in buffer frame `0b011`. Then, this page is latched and its frame address is returned.

The following call of FIXPAGE(**6**, SH) does not work like the preceding ones. The call of HASHTABLE.LOOKUP(**6**) returns *NIL*, which signals a page miss. As the buffer pool is currently full, a page needs to be evicted. Assuming that the eviction strategy picks the page[2] with page ID **5** to be evicted from buffer frame `0b001`. Therefore, it is exclusively latched and removed from the hash table. Such a removal is hardly more expensive than a search inside it. The most expensive task during a page miss is the subsequent retrieval of the page by the storage manager, which is then stored in the buffer frame with index `0b001`. The mapping from page ID **6** to buffer frame index `0b001` is added calling HASHTABLE.ADD(6, `0b001`), which is roughly as expensive as a search in the hash table. As the requested latch was of type SH, the exclusive latch held by the calling thread gets downgraded to that type and the page is returned.

The calling transaction can now find the searched key 250 in that page.

[2]Depending on the chosen page replacement algorithm, the selection of a victim can be very costly.

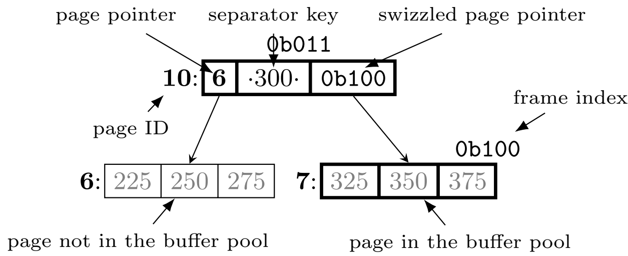

Figure 2: Notation in Figures 1 and 5

2.3 Cost of this Technique

As shown in Figure 4, the lion share of the time spent during a page miss is required for the eviction of the victim page and the retrieval of the requested DB page, in this case on an SSD. On a system running 8 worker threads concurrently, the majority of time spent during a page hit is waiting for the latch. This considers the average case only, as most page fixes do not need to wait to acquire the latch. When only those page fixes are considered where a shared latch was requested, the average time required for latching drops by ∼30 %. But the hash table lookup also requires a significant time share during a page hit—in such a case, it is responsible for ∼15 % of the cost. Section 3 below describes how to mitigate this cost.

When comparing the overall time required for a page fix, a page miss takes in average 19× longer than a page hit. Figure 3 illustrates the overhead due to buffer management for different hit rates. When the hit rate is at 75 %, more than 85 % of the entire runtime of page fixes is required for the remaining 25 % of page misses. Therefore, it is more important to optimize the performance of page misses when the available main memory is smaller[3]. But when the whole working set fits into the buffer pool, then the hit rate will be close to 100 %; for example, for a hit rate of 95 %, the page hits already require around 50 % of the runtime of the buffer manager. Therefore, the overhead due to buffer management would be significantly decreased if the hash table lookup is omitted.

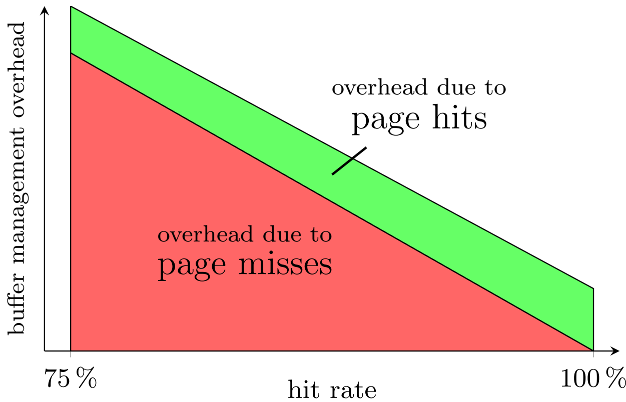

Figure 3: Overhead imposed by the buffer manager represented by the total time spent for fix pages

[3]The higher I/O latency of HDDs increases the runtime of page misses, which is another reason why the optimization of page hits was not that effective years ago.

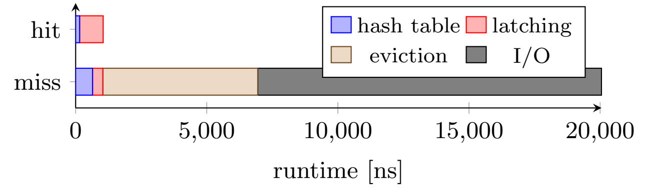

Figure 4: Runtime of an average page fix: Illustration of time slices for the tasks executed[4].

3. USAGE OF POINTER SWIZZLING

3.1 Fix a Page

The fixing of a DB page when using pointer swizzling requires the same basic tasks as the traditional technique presented in the previous section. But due to swizzling, the parent page is required during page misses. To keep the buffer pool with pointer swizzling simple, it requires that only one pointer per page referencing it should exist (Figure 5). Therefore, a B+tree index where the data pages are not only linked to the inner navigational pages but also to their following (and preceding) siblings to form a linked list [2], is not supported. In the original work [3], the write-optimized Foster B-tree was used [4]. The restriction to one pointer per page makes it simple to guarantee that every pointer to a page in the DB buffer is always swizzled. This makes the management of swizzled pointers during page misses and during page eviction much simpler. Another consequence of that restriction is the fact that a transaction, before fixing a page, needs to fix its parent page and therefore it can offer the frame ID of the parent page with nearly no overhead when fixing a page. This makes the swizzling of the pointer inside that page much cheaper.

To facilitate the comparison between the new approach and the traditional technique, fixing of a page inside a buffer pool using pointer swizzling is presented in Algorithm 2 in the same way the traditional approach was presented in Section 2.

The very first step when fixing a page under pointer swizzling buffer management (PSBM for short) is a check if the page ID is swizzled, which is done on line 2. To that end, the most significant bit in the page ID format is used. The overhead of one bit does not significantly restrict the maximum number of pages the DBMS can manage—in fact, most 64-bit CPU architectures only use 48 bits for virtual memory addresses, and the unused bits are commonly used for techniques similar to pointer swizzling. If a page pointer is swizzled, the pointer contains the frame index where the requested page can be found (line 3) instead of its page ID.

If a pointer is not swizzled, i.e., it contains the actual page ID, the page is not in the buffer. To bring the database page into the buffer pool, the steps required in the traditional approach need to be taken here as well on lines 6 to 8, 14, 15, 17 and 18. But instead of updating the hash table, the pointer in the parent page of the new page needs to be swizzled and in case of a full buffer pool, the pointer in the parent page of the evicted page needs to be unswizzled.

To unswizzle a pointer inside the in-memory copy of the parent page of the replacement victim, the frame where this page can be found needs to be located. The arguments of the FIXPAGE function only contain the buffer frame index where

[4]Minor management tasks omitted.

```
1: function fixPage(pageID, latchType, parentFID)
2:   if swizzled(pageID) then                              ▷ Hit
3:     frameID ← pageID
4:   else                                                  ▷ Miss
5:     if full then
6:       frameID ← chooseVictim
7:       acquireLatch(frameID, EX)
8:       victimPID ← getPageID(frameID)
9:       victimPaFID ← getParent(frameID)
10:      acquireLatch(victimPaFID, EX)
11:      unswizzle(victimPaFID, frameID, victimPID)
12:      releaseLatch(victimPaFID)
13:    else
14:      frameID ← chooseFreeFrame
15:      acquireLatch(frameID, EX)
16:    end if
17:    page ← storageManager.retrieve(pageID)
18:    insertIntoFrame(frameID, page)
19:    setParent(parentFID)
20:    acquireLatch(victimPaFID, EX)
21:    swizzle(parentFID, pageID, frameID)
22:    releaseLatch(victimPaFID)
23:  end if
24:  acquireLatch(frameID, latchType)
25:  referenceStatistics(frameID, pageID)
26:  return get(frameID)
27: end function
```

Algorithm 2: Page fix with pointer swizzling

the parent page of the requested page can be found, but as the replacement victim is picked independently from the calling transaction, it is not known at the time when the fixPage function is called. Therefore, locating its parent page needs to be done differently. When fixing a page, the frame index of its parent page is always known and, therefore, storing this frame index inside the buffer frame of its child page does not introduce significant overhead. Thus, the required frame index of the victim's parent page can be retrieved from there on line 9. But as our implementation still requires the hash table for management tasks, the frame index of the parent page is maintained there. To perform unswizzling—which actually implies a change of the parent page—thread safe, the parent page is latched in exclusive mode on line 10. During unswizzling, the page reference field inside the page buffered in frame victimPaFID containing frameID is replaced with the victimPID on line 11. Afterwards neither the replacement victim nor its parent is required anymore during this page fix and therefore the latch of the victim's parent can be released. The latch of the frame, where the replacement victim was found, cannot be released because this frame is required to buffer the requested page in it.

If the buffer pool was not full when a page miss happened, an unused frame needs to be chosen on lines 14 and 15 and this task does not interfere with locating a page. Therefore, it is the same as in the approach using a hash table.

The retrieval of the requested page from persistent storage (lines 17 and 18) also works the same as before. But while the page was added to the hash table at this point of the traditional approach, here swizzling is required. To support unswizzling as described before, the frame index of the parent page is stored in the frame of its child page on line 19. The swizzling step on line 21 works similarly to unswizzling: the page pointer inside the parent frame is replaced with the frame ID, with the swizzled bit set to 1. And as the replacement of pointers changes that parent page, its latch needs to be acquired in exclusive mode (line 20) just for that task (line 22).

Once the frame is properly located and loaded, it is latched in line 24. Like in the traditional approach, this potentially downgrades the exclusive latch, if it is a page miss. The notification of the page reference (line 25) and the return of the reference to the page fixed (line 26) also needs to be done the same way as it is done in the traditional approach.

3.2 B+Tree Traversal

The initial state of the B+Tree and of the buffer manager in Figure 5 is the same as in Section 2. But instead of having a hash table as additional data structure, some of the page pointers in the pages **1**, **5** and **10** are swizzled.

When a record with key 250 has to be accessed, the search in the B+tree starts at the root page. The page ID of the root—**1**—could be found in the system catalogs in the same way as via hash table access. But when pointer swizzling is used between the pages of an index structure, it can also be applied to the pointers from the system catalogs to the root pages of the index structures used. Therefore, the system catalogs contain the buffer frame indexes—here `0b000`—of root page currently kept in the buffer. Furthermore, a root page can be fixed using that buffer frame index and using a simplified version of Algorithm 2.

The next page on the access path to the record with key 250 has page ID **10**. But as the pointer between separator keys ·200· and ·400· contains the swizzled pointer `0b011`, the next call to the buffer manager will be fixPage(`0b011`, SH). For example, using the technique to recognize swizzled pointers introduced above[5], a call to swizzled(`0b011`) would return *true*. Hence, the buffer manager recognizes a page hit and, in turn, it treats `0b011` as the buffer frame index where the requested page **10** can be found. The only important steps left to fix that page are to acquire its latch calling acquireLatch(`0b011`, SH) and to return the actual page address by executing **return** get(`0b011`).

As the next page on the access path with page ID **6** is not currently held in main memory, the pointer before separator key ·300· is not swizzled and therefore just contains the page ID **6**. The following call of fixPage(6, SH) therefore does not work like the preceding calls to that function. The call of swizzled(6) returns *false*, which signals a page miss[6]. As the buffer pool is currently full, a victim page needs to be chosen. Assuming, the page eviction strategy picks the page with page ID **7** to be evicted from buffer frame `0b100`. In the example in Section 2, the page eviction strategy picked page **5** for eviction but as there are child pages of that page buffered in the buffer pool (pages **2** and **3**), this page cannot be evicted when using pointer swizzling. To evict the page in buffer frame `0b100`, the frame gets exclusively latched and the page ID **7** of the contained page gets read[7] from the buffer frame. The call of getParent(`0b100`) returns `0b011` and, as this buffer frame is already latched, the call of acquireLatch(`0b011`, EX) will only upgrade the shared latch to exclusive mode. With that page latched exclusively, the

[5]The bit of the page ID dedicated to that recognition is not shown in this example.

[6]Implied by the fact that the parent page does contain an actual page ID instead of a swizzled pointer.

[7]The pickVictim function could also directly return that.

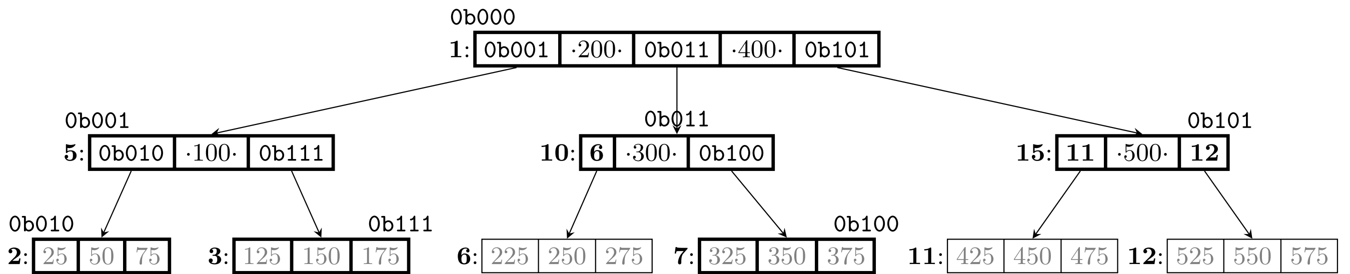

Figure 5: The same data structure and data like in Figure 1: The buffer manager uses pointer swizzling to locate pages.

swizzled page pointer `0b100` can be replaced by the page ID **7**. This unswizzling task is executed by calling UNSWIZZLE(`0b011`, `0b100`, 7). The latch of the changed page **10** can now be released as the eviction of its child page has completed. But as that page is also the parent page of page **6**, it was already latched in shared mode before the eviction[8] and therefore RELEASELATCH(`0b011`) only downgrades the latch to the mode (shared) before the call of ACQUIRELATCH(`0b011`, EX). Now that the buffer frame `0b100` is unused, page **6** can be retrieved from secondary storage using the storage manager and buffered in that location. Using pointer swizzling, the pointer to a page brought to the buffer pool needs to be swizzled. Therefore, call SWIZZLE(`0b011`, 6, `0b100`) replaces in page **10** the pointer in ahead of of the separator key by the buffer index `0b100`. To support the unswizzling of that pointer required during the eviction of that page **6**, the buffer frame index `0b011` gets stored in a dedicated field in buffer frame `0b100`. As a consequence, the swizzled pointer can be easily found when frame `0b100` is picked for eviction. The address of the page stored in buffer frame `0b100` can now be returned to the calling transaction as this frame now contains the requested page **6**.

The calling transaction can now locate the searched key 250 in that page.

3.3 Cost of this Technique

Compared to the traditional buffer management (TBM) approach, a page hit does not require a hash table lookup; therefore, the only cost of a page hit for PSBM is the time for latch acquisition, as presented in Figure 6. But this overhead is identical for both approaches and, hence, average page hits are ~15 % faster due to pointer swizzling[9].

For page misses, the only differences between both approaches apply to page eviction and the swizzling procedure. As we use RANDOM page replacement[10] in our experimental DBMS, the differences in hit rate and runtime of page eviction between the two approaches are insignificant. However, pointer swizzling contributes some overhead to page misses. However, the swizzling step adds a small overhead of ~2.5 % to page misses.

Not included in the algorithmic sketch of PSBM is the usage of a hash table for additional management tasks such as synchronizing concurrent reads and writes on persistent storage, i.e., managing in-flight I/Os.

[8] And it will also be required when the page pointer to page **6** gets swizzled.

[9] Compare the page hits in Figure 4 with those in Figure 6

[10] With RANDOM replacement, CHOOSEVICTIM selects a random page not in use and not containing swizzled pointers.

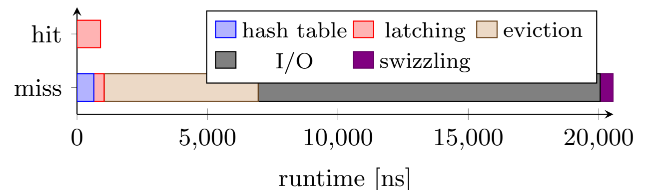

Figure 6: Runtime of an average page fix for PSBM: Illustration of time slices for the different tasks executed[11].

During the runtime measurements of different phases of page fixes, we encountered a great spread of runtimes for some of the phases. It is clear that some page fixes wait very long for a latch (exclusive latching of pages in higher levels of the B+Tree), while others instantly acquire the requested latch (page misses imply that the pages were not in use). We also note that the average time required for page fetching drastically varies (~10 000 ns–17 500 ns) for the SSD used in the experiments between multiple runs of the same benchmark—this could be attributed to variations in the internal flash translation layer and garbage collection. More precisely, the runtime of hash table operations, swizzling, and latching during page misses were the only measured values that do not show up significant variability (standard deviation of <4.5 % of the average value). The standard deviation of the other phases was at 14 %–23 % of the average measured value.

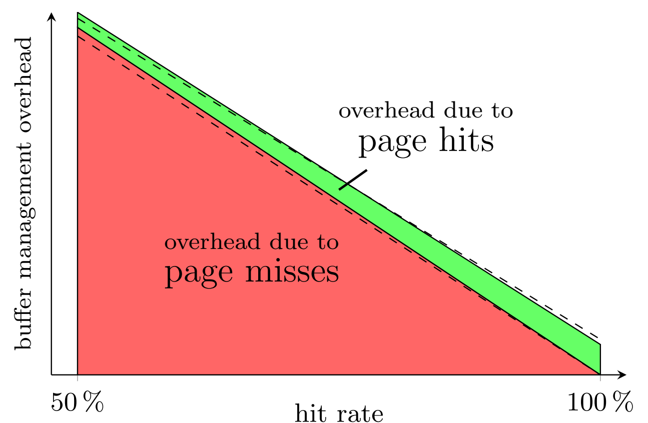

Figure 7: Overhead imposed by the buffer manager represented by the total time spent to fix pages. The time spent for page misses also includes the overhead due to the underlying system layers (incl. I/O latency).

[11] Minor management tasks omitted.

The total runtime of page hits and page misses for different hit rates is presented in Figure 7. The dashed lines show the comparison with TBM. It shows that page misses are slightly cheaper in the traditional approach while page hits are more expensive there. Therefore, PSBM can profit from high hit rates while the usage of TBM is an advantage when the hit rates are lower. While both buffer management techniques obviously profit from higher hit rates, our measurements clearly show superior results of PSBM for hit rates >75 %.

4. PERFORMANCE EVALUATION

The evaluation of our implementation of PSBM[12] using the TPC-C benchmark is shown in Figure 8. The experiments were performed on a system equipped with an *Intel®Xeon® E5420* CPU, 32 GB of RAM and three 256 GB *Samsung 840 PRO* SSDs. Transactional log and database file were stored on separate SSDs, TRIMed before each benchmark run.

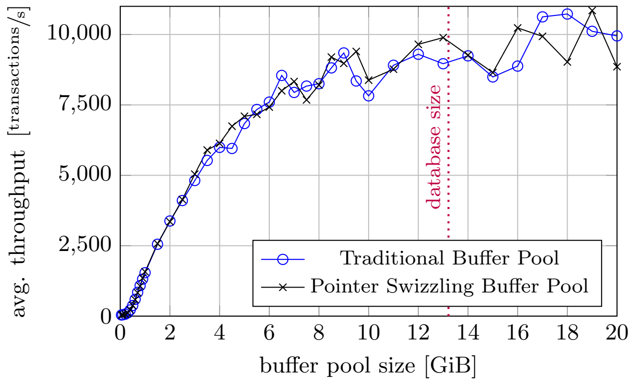

Figure 8: TPC-C transaction throughput for different buffer pool sizes and a database of 100 warehouses ($\implies$ ∼13.2 GB database size) queried by 8 terminals

Using micro-benchmarks, our measurements inside the buffer manager have led to the observation that pointer swizzling improves the performance for high hit rates and, in turn, also for larger buffer pool sizes. But the evaluation of the overall DBMS does not confirm this observation.

Even when the whole database fits in the buffer pool, the performance of PSBM is not better than the performance of TBM. It would be expected that a hit rate close to 100 %[13] leads to the performance advantage due pointer swizzling as measured inside the buffer manager. We believe this can be attributed to the presence of other bottlenecks in the system architecture, and further measurements will be performed in future work to investigate this issue.

5. CONCLUSIONS

The performance of many applications caching persistent objects suffers from the overhead imposed by the mapping of persistent addresses to memory addresses. To address that overhead, pointer swizzling was introduced initially to so-called *Load-Work-Save* applications, where a batch of persistent objects (typically representing a self-contained project) is loaded into an object cache to work on those objects. After fetching the objects, the persistent pointers between those objects are replaced with the memory addresses of the transient copies. When having finished working on a batch of objects, the transient copies are saved persistently. But before, the persistent pointers are restored as the memory addresses will become invalid once the objects are removed from the object cache. This technique is called eager pointer swizzling (classification proposed in [6]).

DB buffer management does not have those clear phases of caching and uncaching of pages, but large buffer pools also lead to situations where almost all page references are page hits and, therefore, the page IDs used by the storage manager are not required. But as database pages are cached on demand and uncached whenever page eviction is required, the usage of direct, lazy pointer swizzling, replacing the page IDs with memory addresses decreases buffer management overhead on page hits.

We could measure those positive effects of pointer swizzling inside the buffer manager, but our implementation does not lead to the expected performance improvement of the overall workload (as measured by transaction throughput). The wide variance of benchmark results leads us to the belief that there are other unexpected bottlenecks in the system architecture that must be addressed. We expect that a proper implementation of PSBM could actually show an advantage due to pointer swizzling.

The runtime measurements inside the buffer manager lead to the observation that higher hit rates increase the performance advantage of pointer swizzling and, hence, the usage of better page replacement algorithms is an obvious idea. Up to now, we implemented and evaluated the CAR [1] algorithm in our buffer manager leading to significantly higher hit rates. We did not use this page replacement algorithm in the context of this work as the high overhead of our implementation of CAR has eaten up the performance advantage due to higher hit rates. Therefore, another goal of future work is optimizing that implementation.

6. REFERENCES

[1] S. Bansal and D. S. Modha. CAR: clock with adaptive replacement. In *FAST '04*, pages 187–200, 2004.

[2] D. Comer. The ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, 1979.

[3] G. Graefe et al. In-memory performance for big data. *PVLDB*, 8(1):37–48, 2014.

[4] G. Graefe, H. Kimura, and H. A. Kuno. Foster b-trees. *ACM Trans. Database Syst.*, 37(3):17:1–17:29, 2012.

[5] S. Harizopoulos et al. OLTP through the looking glass, and what we found there. In *SIGMOD '08*, pages 981–992, 2008.

[6] S. J. White and D. J. DeWitt. Quickstore: A high performance mapped object store. In *SIGMOD '94*, pages 395–406, 1994.

[12] Our implementation is based on *Shore-MT*.

[13] Initially the DB buffer needs to warm up but the effect of this is insignificant when the benchmark runs for 50 min.