TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Master's Thesis

Bottlenecks Uncovered: A Component-Wise Breakdown of the Runtime of an OLTP System

by **Max Fabian Gilbert***

Day of Issue: February 1, 2020
Day of Release: September 14, 2020

Advisor: M. Sc. Caetano Sauer
First Reviewer: Prof. Dr.-Ing. Stefan Deßloch
Second Reviewer: Prof. Dr.-Ing. Dr. h. c. Theo Härder

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science

in the

Heterogenous Information Systems Group
Department of Computer Science

*m_gilbert13@cs.uni-kl.de

Declaration of Authorship

I, Max Fabian Gilbert, declare that this thesis titled, “Bottlenecks Uncovered: A Component-Wise Breakdown of the Runtime of an OLTP System” and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.

Kaiserslautern, September 14, 2020 Max Fabian Gilbert

This page intentionally left blank.

Zusammenfassung

In der Vergangenheit war die Reduzierung der Plattenzugriffe für die Leistung eines DBVS entscheidender als die Reduzierung der für eine bestimmte Aufgabe erforderlichen CPU-Zyklen. Es war wichtig, die CPU mit der Bearbeitung anderer Transaktionen auszulasten, während eine Transaktion darauf warten musste, dass benötigte Daten von der Festplatte geladen werden. Mit stetig wachsendem Arbeitsspeicher in modernen Servern und der verbesserten Leistung von NVRAM-Produkten wurden Hauptspeicher-DBS mehr und mehr zur Norm. In diesen Systemen sind Optimierungen für HDD- — oder heute SSD- — Zugriffe für die reguläre Transaktionsverarbeitung viel weniger wichtig.

Ohne diesen leistungseinschränkenden Faktor wurde die Minimierung der zur Ausführung einer Transaktion erforderlichen CPU-Zyklen zum Schlüssel für die Leistungsmaximierung von DBS. Um die Motivation für ein komplettes Redesign von DBVS zu liefern, identifizierten Harizopoulos et al. in [Har+08] verschiedene Komponenten des *Shore Storage Manager*, die die überwiegende Zahl der CPU-Zyklen während der Ausführung einiger Transaktionen des *TPC-C* Benchmarks in Anspruch nehmen. Sie kamen zu dem Schluss, dass die CPU auf modernen Systemen nur in einem sehr kleinen Teil — 1 %–2 % — der Zeit nützliche Arbeit leistet. In Kapitel 2 dieser Arbeit wiederhole ich deren Messungen mit einer anderen Methodik an einem modernisierten Nachfolger des *Shore Storage Manager*.

Allerdings ist die Ausführung von OLTP-Anwendungen auf schwacher Hardware noch immer im privaten und kommerziellen Bereich zu finden, und es werden auch dort stetig neue Arbeitslasten hinzugefügt, so dass DBVS-Optimierungen, die sich auf die Minimierung des HDD- oder SSD-Zugriffs konzentrieren, immer noch Sinn ergeben. Aus diesem Grund werden in Kapitel 1 viele gängige Seitenersetzungsalgorithmen untersucht, die im Laufe der Geschichte vorgeschlagen wurden. Es reicht jedoch nicht aus, in einer Leistungsbewertung dieser Seitenersetzungsalgorithmen nur die Reduzierung von SSD-Zugriffen zu betrachten, da einige von ihnen das Potenzial haben, selbst zu einem Flaschenhals für ein DBS zu werden. Aus diesem Grund wurden alle diese Algorithmen für ein echtes DBVS implementiert und im Hinblick auf Fehlseitenrate und Transaktionsdurchsatz bei *TPC-C* verglichen.

Für meine Bachelorarbeit hatte ich die Pointer Swizzling für den Pufferpool, wie von Graefe et al. in [Gra+14] vorgeschlagen, bereits neu evaluiert. Aufgrund von Stabilitätsproblemen des verwendeten DBMS-Prototypen waren die Messungen jedoch kaum aussagekräftig. Eine Neuevaluierung dieser Technik ist im Abschnitt 2.4.1 zu finden.

This page intentionally left blank.

Abstract

Traditionally, reducing disk accesses has been more decisive to the performance of an OLTP system than reducing the number of CPU cycles needed for a given workload. It was important to keep the CPU busy—executing other transactions—while transactions had to wait for data to be fetched from hard disk. With the ever-increasing amount of available RAM in modern servers and the improved performance of NVRAM products, in-memory OLTP systems became more and more the norm. In these systems, optimizations for disk—or today SSD—accesses are much less important for regular transaction processing.

Without this performance constraining factor, minimizing the CPU cycles required to execute a transaction became the key to maximizing OLTP system throughput. To provide motivation for a complete redesign of DBMSs for these new conditions, Harizopoulos et al. identified various components of the *Shore Storage Manager* that take the vast majority of CPU cycles during the execution of some transactions of the *TPC-C* benchmark [Har+08]. They concluded that the CPU does useful work on these modern systems only in a very small portion—1 %–2 %—of the time. In Chapter 2 of this thesis, I repeat their measurements using a different technique on a modernized successor of the *Shore Storage Manager*.

However, OLTP systems on poor hardware are still found in private and commercial applications, and they continue to be stressed with ever new workloads, so there are still opportunities for DBMS optimizations that focus on minimizing HDD or SSD access. For this reason, Chapter 1 assesses many common page replacement algorithms that have been proposed throughout history. However, it is insufficient to address only the reduction of SSD accesses in a performance evaluation of these page replacement algorithms, as some of them have the potential to become a bottleneck for a DBS. For this reason, all these algorithms have been implemented for a real DBMS and compared in terms of hit rate and *TPC-C* transaction throughput.

For my Bachelor's thesis I re-evaluated the Pointer Swizzling technique for the buffer pool that Graefe et al. proposed in [Gra+14]. However, due to stability problems of the DBMS prototype used, the measurements were hardly meaningful. A re-re-evaluation of this technique can be found in the Section 2.4.1.

This page intentionally left blank.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

1 Page Eviction by the Buffer Pool Management

1.1 DBMS Buffer Management

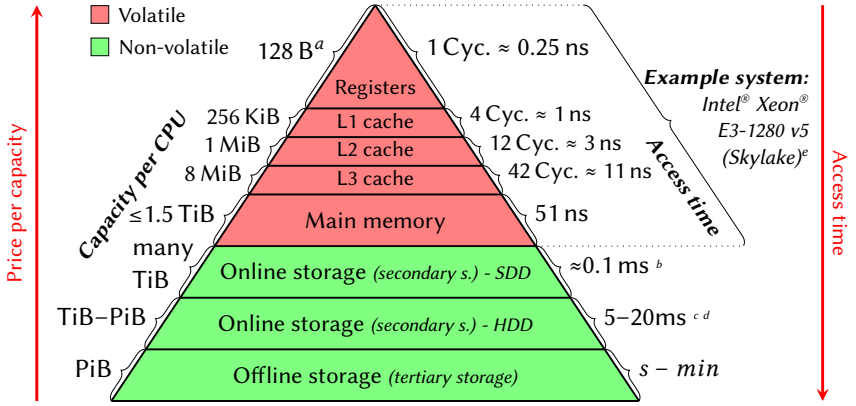
The buffer management of a typical disk-based DBMS serves the purpose of providing the higher layers of the DBMS with managed access to data and metadata pages of a database stored in files on the secondary storage. This managed access involves fetching specific pages into specific memory locations—so-called *buffer frames*—inside the buffer pool and write-back changes to pages in memory to database files in secondary storage.

Today, there are server systems with 48 TiB of cache-coherent shared memory[1] that allow *in-memory* management of almost any database of any application. But such expensive systems (>2 000 000 €) do not pay off for most applications even with large databases. Therefore, there will still be many situations in which the main memory of a system is significantly smaller than the database(s) managed on it. Accordingly, the number of buffer frames in the buffer pool is then smaller than the number of pages in the database.

Therefore, the buffer management must *evict* pages from buffer frames if currently not buffered pages are referenced while there are no more free buffer frames. For this purpose, each buffer manager has a page eviction module—implementing one of the many *page replacement algorithms* developed since the 1960s.

In *OLTP* (online transaction processing) applications, the majority of database accesses are random accesses, so the access latency of the underlying storage technology (figure 1.1) is critical to performance. The main goal of buffer management is to maximize the *hit rate* in the DB buffer by

[1] https://h20195.www2.hpe.com/v2/gethtml.aspx?docname=c04912781



[a] Only general purpose registers are considered
[b] https://www.samsung.com/semiconductor/global.semi.static/SSD_vs_HDD_Brochure_181001.pdf
[c] https://web.archive.org/web/20180509053527/http://www.tomshardware.de:80/enterprise-hdd-sshd,testberichte-241390.html
[d] https://www.computerbase.de/2016-08/seagate-enterprise-capacity-3.5-hdd-10tb-test/
[e] https://www.7-cpu.com/cpu/Skylake.html

Figure 1.1: Storage hierarchy of computer systems

keeping as many pages of the working set (pages referenced in the near future) as possible in relatively fast (51 ns) main memory to avoid expensive (>100 µs) secondary storage accesses.

Bélády's optimal page replacement algorithm [Bél66] evicts the one page from the buffer pool that is *not re-referenced for the longest time in the future*[2]. However, this algorithm cannot make this decision at runtime, since this would require knowledge about future requests to the database. Therefore, page replacement algorithms that can be implemented in DBMS must use *heuristics* to approximate the eviction decisions of Bélády's algorithm to achieve a high hit rate. These heuristics are usually based on the assumption that there is a temporal locality of the page references. The management of data in B-tree indices, which were developed 50 years ago to improve the spatial locality of references in order to reduce random

[2] The more practical 5 Minute Rule [GP87] which is derived from this algorithm is still valid [GG97][Gra07].

disk access in database applications [BM70], increases the temporal locality of page references due to the hierarchical structure of B-trees in which higher-level pages are more likely to be referenced.

The expected performance of these possible page replacement algorithms relative to the buffer pool size is shown in figure 1.2. In this diagram, MR_{CS} is the minimum miss rate, which is greater than 0 due to the page misses that occur when each page is first referenced after a cold start. The maximum miss rate $MR_{\max}$ is less than 1 due to the fact that there are always some random page hits, even with RANDOM eviction on workloads with no locality of references. D is the database size and $B_{\min}$ is the minimum buffer pool size that will still lead to a working system.

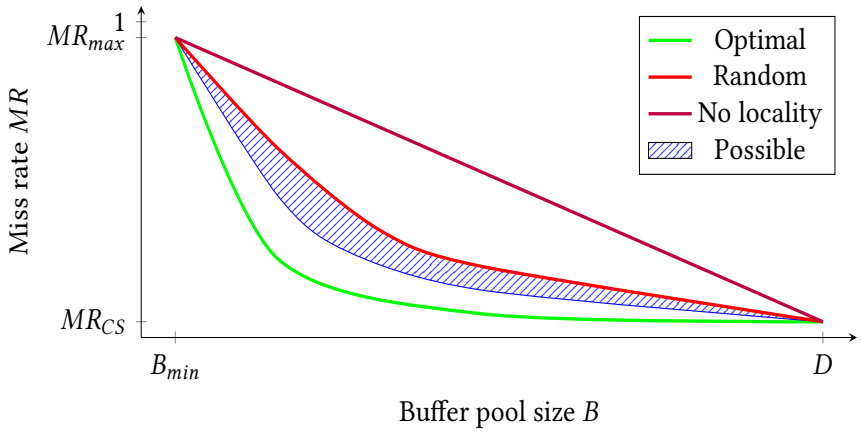


Figure 1.2: Connection between miss rate and buffer pool size [EH84]

Besides the impossibility to implement Bélády's optimal page replacement algorithm, there are also technical limitations in the use of the possible page replacement algorithms. The abstract algorithms can *evict any page* from the buffer pool *at any time*. However, in a *real-world system*, there are many reasons why a page that has been selected for eviction by the page replacement algorithm *cannot be immediately evicted*. One reason, for example, is already visible in the typical interface of a buffer pool—a page can be *fixed* and *unfixed* by the higher layers of the DBMS. Between fixing and unfixing a page, it is guaranteed that it remains in the same buffer frame since the fixing thread processes the page during this time. Another problem for some page replacement algorithms is the possibility to

explicitly evict pages from the buffer pool—independent from the decisions of the page replacement algorithm.

The *test system* used for this work knows the following reasons why a page in the buffer pool is either temporarily or permanently unreclaimable:

- Metadata pages can never be evicted.
- B-tree root pages can never be evicted.
- Inner B-tree pages can never be evicted when pointer swizzling (like in [Gra+14]) is used in the buffer pool.
- B-tree pages with foster children (the Foster B-Tree from [GKK12] is used) cannot be evicted.
- Dirty pages cannot be evicted until they are written back.
- Pages pinned to the buffer pool by higher layers of the DBMS cannot be evicted.

All non-trivial page replacement algorithms collect statistics about page references to base their eviction decisions on. Simple implementations of these page replacement algorithms do not reflect in their statistics the fact that certain pages are temporarily or permanently unreclaimable. But treating pages found temporarily unreclaimable as a page reference and excluding pages that are permanently unreclaimable from the eviction could improve future eviction decisions or the runtime of the algorithm.

1.2 Page Replacement Strategies

There are two traditional classifications for page replacement algorithms—Bélády's classification from [Bél66] and the classification by Effelsberg and Härder from [EH84].

Class 1 These replacement algorithms do not keep any statistics.

Class 2 These replacement algorithms keep statistics about the latest references of *pages in the buffer pool* and use those for their decisions.

Class 3 These replacement algorithms keep statistics about each time *any page* was fetched from the database and each time it was evicted from the buffer pool and use those for their decisions.

Table 1.1: Bélády's classification of replacement algorithms from [Bél66]

Bélády grouped the replacement algorithms into the *three classes* described in table 1.1. The RANDOM, FIFO and FILO page replacement algorithms belong to Class 1, the modern (proposed after 2002) ARC [MM03], CAR [BM04], CART [BM04], LIRS [JZ02], CLOCK-Pro [JCZ05], DLIRS [Li18] and CLOCK-Pro+ [Li19] page replacement algorithms belong to Class 3. All the other page replacement algorithms (LRU, MRU, LRU-K [OOW93], SLRU [KLW94], CLOCK [Cor69], ZCLOCK, GCLOCK [EH84], DGCLOCK [EH84], LRD [EH84], LFU, LFU-Aging [AFJ00], LFUDA [Arl+00], 2Q [JS94], MQ [ZPL01] and LeanStore [Lei+18]) belong to Class 2.

The classification by Effelsberg and Härder is *two-dimensional* as it takes into account if a page replacement algorithm considers the time since a certain reference of a page happened—the *age*—and the number of references of a page—the *references*. While this classification is *very detailed*, it only covers the page replacement algorithms from Class 1 and Class 2 of Bélády's classification. Table 1.2 shows each page replacement algorithm covered in this thesis, classified in an extended version of the classification by Effelsberg and Härder.

Consideration during selection decision		**Age**			
		No consideration	Since most recent reference	Since some recent reference	Since first reference
References	No consideration	RANDOM			FIFO FILO
	Most recent reference	ZCLOCK	LRU MRU CLOCK GCLOCK-V2 DGCLOCK-V2 LeanStore		
	Some recent references		SLRU	LRU-K LRD-V2	
	All references	LFU	GCLOCK-V1 DGCLOCK-V1		LRD-V1 LFUDA

Table 1.2: Extended page replacement algorithm classification by Effelsberg and Härder from [EH84]

1.2.1 RANDOM

Introduction RANDOM eviction is the *simplest page replacement policy* because it does not keep statistics on past page references—just a random page from the buffer pool is selected for eviction.

In the course of the *history* of computer science many pseudo-random number generators (PRNG) have been developed. A performance evaluation of *98 PRNGs* related to DB buffer page replacement algorithms in my Project Thesis [Gil20] revealed that each PRNG is suitable for implementing a RANDOM page replacement algorithm. The evaluated PRNGs lead to *identical hit rates* and therefore the one with the *lowest overhead* per generated random number—that is the index of the buffer frame that is reclaimed—should be chosen. Therefore, *SplitMix32* is used for the performance evaluation in section 1.3.

Advantages The main advantage of the RANDOM page replacement policy over the Class 2 and Class 3 competitors is that it does not cause *overhead* for page hits, while the more complex page replacement algorithms always have to update their page reference statistics. In situations where hit rates are high nevertheless, such as when the entire database fits in the buffer pool—which is not uncommon in modern database systems—RANDOM eviction is the *fastest page replacement policy*.

Disadvantages For the RANDOM page replacement policy, none of the pages in the buffer pool are more likely to be referenced again than any other page in it. This results in poor hit rates—compared to more sophisticated page replacement algorithms—but due to the locality of the page references, the hit rate is not even close to 0 %, because, in general, the pages in the buffer pool are more likely to be referenced than the pages that are currently not in the buffer pool.

It is, for example, possible that the RANDOM page replacement algorithm selects a B-tree root page, normally located in the DB's working set, for eviction—the possibility is small because there are usually millions of buffer frames in a DB buffer pool, but only a few B-tree indexes—each with only one root page. But due to the fact that a B-tree root page must be fixed for each search in its corresponding B-tree, it is more likely that these pages are fixed at a certain point in time—where they are then not eligible for eviction—than any other page, even though they usually contain fewer keys than the inner B-tree pages and are therefore fixed for a shorter time per search access.

1.2.2 First In, First Out (FIFO)

The FIFO page replacement policy always evicts the *oldest page first*—it organizes the buffer frames in a *FIFO queue*, which is usually implemented as a circular buffer. The idea behind FIFO page eviction is that the more recently fetched pages are more likely to be referenced again than pages fetched a long time ago. An obvious *counterexample* against the optimality of this heuristic is the B-tree root, which is accessed first when searching in a B-tree, but which has the highest probability of being referenced again.

1.2.2.1 LOOP

Introduction FIFO page eviction can be easily implemented in the buffer pool by arranging the buffer frames (0–15 in the example) in a *circular buffer*, as shown in figure 1.3. The oldest page in the buffer pool, P_{58} in the example, is at the head of the FIFO queue and is therefore evicted next. The last page fetched is P_6 in buffer frame 14. After P_58 is evicted, the buffer frame 15 contains the newest page in the buffer pool and is therefore the tail of the FIFO queue. The oldest page in the buffer pool is then P_{31} in buffer frame 0 which is now the head of the FIFO queue.

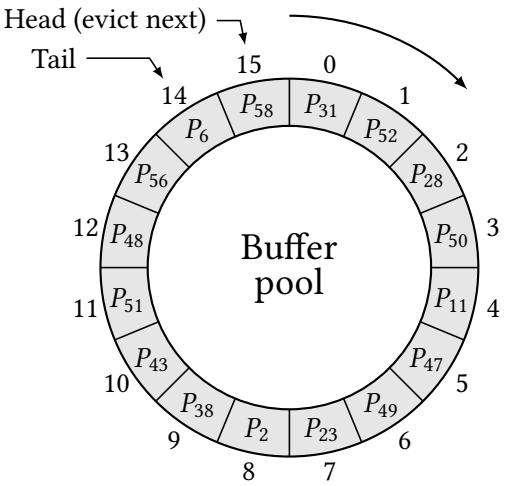


Figure 1.3: The buffer pool arranged as a circular buffer

Page Eviction The page eviction module of the buffer manager therefore simply reclaims buffer frames based on a cyclic counter—implemented e.g. by taking the modulo of a simple infinite counter. This counter-based implementation—*LOOP*—is therefore comparable to a RANDOM page replacement algorithm with a *trivial "pseudo-random" number generator*:

- State word initialized with 0 (seed)
- Next state generated by simple increment
- Identity function is used to map the state word to the returned "pseudo-random" number

If it is temporarily or permanently *not possible to evict* the page at the head of the FIFO queue, the LOOP page replacement algorithm continues iterating over the circular buffer and suddenly places the oldest page in the buffer pool at the tail of the FIFO queue. This leads to a significant deviation from the algorithmic idea of the FIFO page replacement policy. If there are many non-reclaimable buffer frames, the actual page replacements are rather random.

An evaluation of different implementations was conducted for my Project Thesis [Gil20]. The very fast *Local Counter* was used for the performance evaluation in section 1.3 because it scored best in the performance evaluation of my Project Thesis. But this variant *does not properly implement the FIFO page replacement policy* because each working thread uses its thread-local counter. As a result, even in a system with only two working threads—if the counter of one of the threads is only 1 behind the counter of the other thread—a thread will evict from the buffer pool a page that was just fetched into it by the other thread.

1.2.2.2 Quasi-FIFO

Introduction The *Quasi-FIFO* implementation now strives to produce a behavior closer to the FIFO page replacement policy. It does not simply interpret the buffer pool as a circular buffer—like the LOOP page replacement algorithm—but use more complex data structures for its page reference statistics (figure 1.4) and more complicated control structures for determining replacement candidates (Algorithm 1.1).

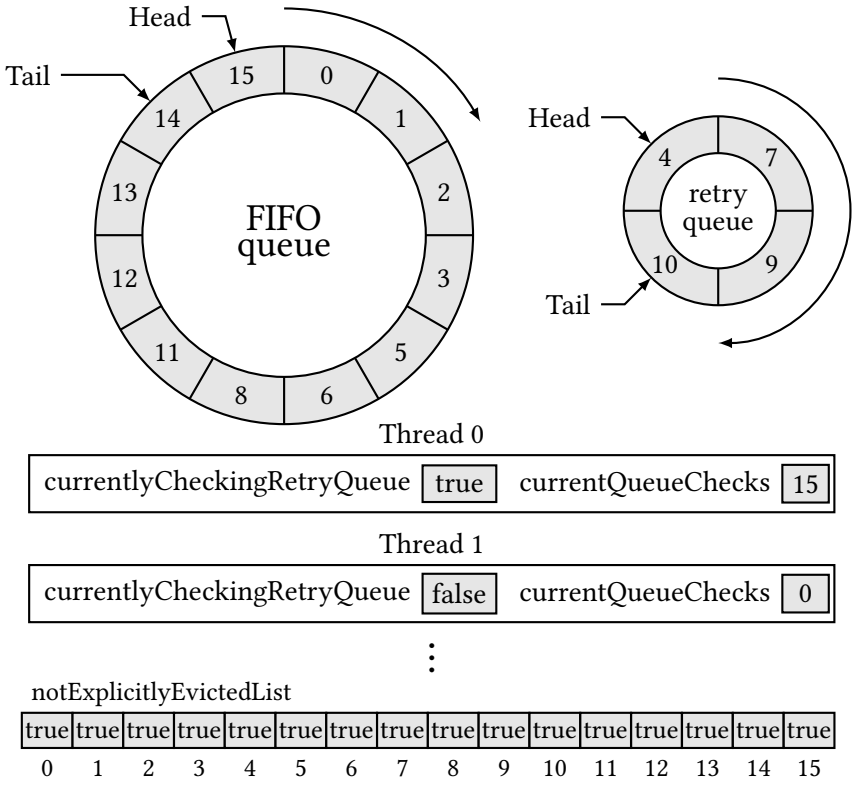


Figure 1.4: The page reference statistics of the Quasi-FIFO page replacement algorithm after almost a complete iteration over the buffer pool

Data Structures As presented in figure 1.4, the page reference statistics of the Quasi-FIFO implementation is composed of two queues, the *FIFO queue* and the *retry queue*, and the notExplicitlyEvictedList array, which is used to flag pages that have been explicitly removed from the buffer pool without involving the page replacement algorithm. The currentlyCheckingRetryQueue flag and the currentQueueChecks counter are used by the threads that evict pages to balance page evictions between the *FIFO queue* and the *retry queue*.

The *example* shows the page reference statistics of the Quasi-FIFO page replacement algorithm operating on a buffer pool with 16 buffer frames—

0–15—, after the events described in this paragraph. Before the first pages were fetched from the database, both the *FIFO queue* and the *repeat queue* were empty. After 16 pages were fetched into the buffer pool, all buffer frame indexes were in the *FIFO queue*—0 at the head and 15 at the tail—and the *retry queue* was still empty since it was not yet necessary to try to evict pages from the buffer pool.

Then thread 0 successfully reclaimed buffer frame 0 and increased its currentQueueChecks counter to 1. Afterward, buffer frame 0 was reused and 0 was added to the tail of the *FIFO queue*. Since the page reference statistics do not contain information about the pages in the buffer frames, the changed ID of the page stored in the buffer frame 0 is not reflected there. Subsequently thread 0 did the same with the buffer frames 1, 2 and 3.

When the buffer frame index 4 was at the head of the *FIFO queue*, it was not reclaimable, and thread 0 moved the buffer index from the head of the *FIFO queue* to the tail of the *retry queue* for later retrying. The page from the buffer frame 5 was then successfully evicted. The figure shows the *state* of the page reference statistics after thread 0 checked 15 pages for eviction (currentQueueChecks) and the pages from buffer frames 0, 1, 2, 3, 5, 6, 8, 11, 12, 13 and 14 were successfully evicted. The buffer frames 4, 7, 9 and 10 were found to be unreclaimable. No retries were made to reclaim a buffer frame that was found to be unreclaimable, and no page was explicitly evicted.

Important to achieve behavior closer to the abstract FIFO page replacement strategy is to balance the eviction of pages from both queues—*FIFO queue* and *retry queue*. This results in the relatively[3] high complexity of the control structures in the **function** SELECT from Algorithm 1.1.

According to the abstract FIFO strategy, the pages in the *retry queue* should have already been evicted from the buffer pool, so the further eviction should *concentrate* on these pages. However, since some of the reasons why a page cannot be evicted temporarily are relatively long-lasting (dirty, foster children) and others indicate that a page is very hot (fixed, pinned), it is more likely that the pages in the *retry queue* can—again—not be evicted, and therefore, too much concentration of the eviction on the *retry queue* would significantly degrade the performance. Pages that

[3] compared to the simplicity of the abstract FIFO page replacement strategy

```
1:  function SELECT
2:      selected ⟵ 0
3:      static currentlyCheckingRetryQueue ⟵ false
4:      static currentQueueChecks ⟵ 0
5:      while true do
6:          if currentlyCheckingRetryQueue == false then
7:              if currentQueueChecks < 0.01 * |FIFOQueue| ∨ |retryQueue| == 0 then
8:                  selected ⟵ FIFOQueue.pop()
9:                  currentQueueChecks ⟵ currentQueueChecks + 1
10:                 if notExplicitlyEvictedList[selected] == true then return selected
11:                 else
12:                     notExplicitlyEvictedList[selected] ⟵ true
13:                     continue
14:                 end if
15:             else
16:                 selected ⟵ retryQueue.pop()
17:                 if notExplicitlyEvictedList[selected] == true then
18:                     currentQueueChecks ⟵ 0
19:                     currentlyCheckingRetryQueue ⟵ true return selected
20:                 else
21:                     notExplicitlyEvictedList[selected] ⟵ true
22:                     continue
23:                 end if
24:             end if
25:         else
26:             if currentQueueChecks < |retryQueue| ∨ |FIFOQueue| == 0 then
27:                 selected ⟵ retryQueue.pop()
28:                 currentQueueChecks ⟵ currentQueueChecks + 1
29:                 if notExplicitlyEvictedList[selected] == true then return selected
30:                 else
31:                     notExplicitlyEvictedList[selected] ⟵ true
32:                     continue
33:                 end if
34:             else
35:                 selected ⟵ FIFOQueue.pop()
36:                 if notExplicitlyEvictedList[selected] == true then
37:                     currentQueueChecks ⟵ 0
38:                     currentlyCheckingRetryQueue ⟵ false return selected
39:                 else
40:                     notExplicitlyEvictedList[selected] ⟵ true
41:                     continue
42:                 end if
43:             end if
44:         end if
45:     end while
46: end function
```

Algorithm 1.1: Selection of eviction candidates by the Quasi-FIFO page replacement algorithm

cannot be evicted permanently should be excluded from the eviction as an *additional optimization* because they would be permanently included in the *retry queue* and therefore, the eviction of these pages would be tried repeatedly, which would reduce the overall performance.

As Algorithm 1.1 reveals, every working thread alternates its eviction activities between the two queues—after trying to reclaim 1 % of the buffer frames in the *FIFO queue*, it tries to reclaim every buffer frame in the *retry queue*. This gives a clear focus on the idea behind the FIFO page replacement strategy and sets the Quasi-FIFO page replacement algorithm off against the LOOP page replacement algorithm.

1.2.3 First In, Last Out (FILO)

The FILO page replacement strategy always evicts the *newest page first*—it arranges the buffer frames in a *FILO stack*. It can therefore be considered the opposite of the FIFO page replacement strategy. This strategy results in very poor performance, because in a buffer pool with n buffer frames, the $(n - 1)$ pages fetched first from the buffer pool remain there permanently, and only the last page fetched is replaced *over and over again*, resulting in a very low hit rate. However, for pages fetched during a table scan, this page replacement algorithm is perfectly suitable as each page is only accessed once, and therefore it could be part of a dynamic page replacement algorithm that uses different page replacement algorithms for different *reference patterns*.

But the implementation of the Quasi-FIFO and FILO page replacement algorithms leads to identical problems with unreclaimable buffer frames, and thus the same solutions to these problems are used in the implementations of both replacement algorithms. In addition to the *FILO stack*, a *retry queue* is used[4]. The algorithm used to select the buffer frames for eviction is almost identical to the one used for the Quasi-FIFO page replacement algorithm (Algorithm 1.1), but instead of the *FIFO queue*, a *FILO stack* is used. However, because the newest page in the buffer pool is usually still fixed, almost all pages selected for eviction are placed in the *retry queue*, degrading this *Quasi-FILO* to FIFO.

[4]The *FIFO queue* from figure 1.4is just replaced with the *FILO stack*.

1.2.4 Least Recently Used (LRU)

The LRU page replacement algorithm improves on the idea of the FIFO page replacement algorithm by considering the time of the last reference to a page, rather than the time it was last fetched from the database file into the buffer pool. It is probably the *most widespread* approach, both in operating system virtual memory management and in DBMS buffer management.

The *intuition* behind the algorithm is that of all pages in the buffer pool, the page that has not been referenced for the longest time in the past (least recently used) will not be referenced for the longest time in the future (Bélády's optimal page replacement algorithm). Although this heuristic is quite intuitive, it is not immune to *suboptimal replacement decisions*. LRU is not scan resistant and prone to page thrashing. The lack of *scan resistance* means that in the case of a table scan, a large number of pages that are usually referenced only once (cold pages) will probably replace all or at least many other pages in the buffer pool—including the hot ones—just because all these pages are used more recently than any previously referenced pages. *Thrashing* refers to a behavior of page replacement algorithms that results in a very low hit rate when the working set of the database is just larger than the buffer pool. For example, if the inner loop of a nested loop join is just one page larger than the buffer pool, LRU will always evict the page used next, resulting in a page fault rate of 100 % (much worse than with RANDOM page replacement), even though a hit rate of almost 100 % could be achieved by sharing only one buffer frame between two pages (with page faults whenever one of these pages is referenced).

1.2.4.1 Hash-Map-Doubly-Linked-List Implementation

Introduction The most straightforward approach for implementing the LRU page replacement strategy uses a *queue* into which each buffer frame index is inserted when a page is fetched into the respective buffer frame. For each page reference, the respective buffer frame index is moved to the tail of the queue. And the buffer frame at the head of the queue is always the one that is reclaimed during eviction.

Page Reference Statistics Because of the requirement to find and remove a specific element (buffer frame index) at any position in the queue (whenever a page reference occurs), an *abstract data type* implementing a simple queue with *enqueue* and *dequeue* operations cannot be used. The *Hash-Map-Doubly-Linked-List implementation* uses a queue implemented using a doubly-linked list inside a hash map (a simple array can be used alternatively), because searching in a bare doubly-linked list would otherwise require traversing all elements from the tail of the queue to the searched buffer frame index, which would require traversing *less than half* of all elements ($\mathcal{O}(n)$) due to the locality of the page references.

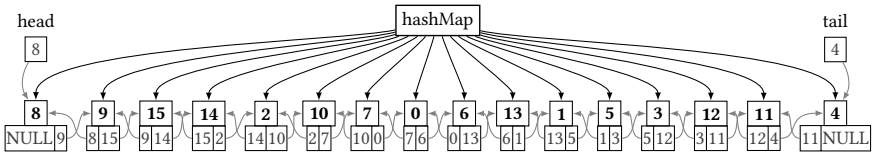


Figure 1.5: Implementation of the LRU queue using a doubly-linked list inside a hash map

Figure 1.5 shows such a double-linked list within a hash map used to obtain a queue with the special properties required for the LRU page replacement algorithm. The hashMap can be any suitable hash map implementation that maps a buffer frame index (e.g., **10**) to its corresponding previous and next pointers (e.g., 2 and 7). The order of the buffer frame indices in the queue (shown in the figure) does not necessarily correspond to the way the hashMap organizes the elements (not specified in the figure). A call to hashMap [10] will return (2, 7).

Updates of the Page Reference Statistics The operations for enqueuing (Algorithm 1.2), dequeuing (Algorithm 1.3) and removing (Algorithm 1.4) a specific buffer pool index are rather trivial, but are nevertheless given here for clarification.

An eviction candidate is selected by dequeuing a buffer frame index from the LRU queue. If it cannot be evicted, it is enqueued again. When a page is fetched into the buffer pool, the corresponding buffer frame index is enqueued in the LRU queue, and when a page hit occurs, the corresponding buffer index is removed from the LRU queue and enqueued again.

```
1: function ENQUEUE(key)
2:     if |hashMap| ≠ 0 then
3:         if hashMap [key] ≠ NULL then
4:             return ERROR!
5:         end if
6:         hashMap [key] ⟵ (tail, NULL)
7:         hashMap [tail] .next ⟵ key
8:         tail ⟵ key
9:     else
10:        hashMap [key] ⟵ (NULL, NULL)
11:        head ⟵ key
12:        tail ⟵ key
13:    end if
14: end function
```

Algorithm 1.2: Enqueue an index into the Hash-Map-Doubly-Linked-List

```
1: function DEQUEUE
2:     if |hashMap| = 1 then
3:         hashMap [head] ⟵ NULL
4:         head ⟵ NULL
5:         tail ⟵ NULL
6:     else if |hashMap| > 1 then
7:         oldHead ⟵ head
8:         hashMap [hashMap [oldHead] .next] .previous ⟵ NULL
9:         hashMap [oldHead] ⟵ NULL
10:    else
11:        return ERROR!
12:    end if
13: end function
```

Algorithm 1.3: Dequeue an index from the Hash-Map-Doubly-Linked-List

A very important part of the implementation has been left out so far. Since there are usually many working threads processing transactions *concurrently* in a modern DBS, there are multiple threads referencing pages concurrently and thus updating the page reference statistics concurrently. For this reason, accesses to the LRU queue must be *synchronized*, which is simply done here via a *global latch* for the entire queue. But such a global latch, acquired during every page fix, severely limits the concurrent transaction processing of the entire DBS and thus becomes a bottleneck of the system. An implementation that scales better with the number of threads is described in the next subsection 1.2.4.2.

```
 1: function REMOVE(key)
 2:     if hashMap[key] ≠ NULL then
 3:         if hashMap[key].next ≠ NULL then
 4:             hashMap[hashMap[key].next].previous ← hashMap[key].previous
 5:         else
 6:             tail ← hashMap[key].previous
 7:         end if
 8:         if hashMap[key].previous ≠ NULL then
 9:             hashMap[hashMap[key].previous].next ← hashMap[key].next
10:         else
11:             head ← hashMap[key].next
12:         end if
13:         hashMap[key] ← NULL
14:     else
15:         return ERROR!
16:     end if
17: end function
```

Algorithm 1.4: Remove an index from the Hash-Map-Doubly-Linked-List

1.2.4.2 Timestamp-Sorting Implementation

Introduction While page hit and page miss overhead is the same for the Hash-Map-Doubly-Linked-List implementation of the LRU page replacement strategy, the Timestamp-Sorting implementation is *optimized for* (on average) fast updates of page reference statistics on *page hits*. As the name suggests, this implementation records the *timestamp of the last page reference* for each buffer frame, and *sorts the buffer frames* by these timestamps in order to evict the least recently referenced page.

Page Reference Statistics Figure 1.6 shows the page reference statistics of this implementation of the LRU page replacement strategy for a buffer pool with 16 buffer frames. The *liveTimestamps* array contains for each buffer frame 0–15 the *timestamp of the most recent reference* to the contained page. The LRUqueue0 and LRUqueue1 arrays are unused until the first buffer frame is reclaimed. But then, either *LRUqueue0* or *LRUqueue1* (indicated by either *useLRUqueue0* or *useLRUqueue1* being true) contains the most up-to-date list of buffer frames (first column), *sorted by the timestamp of references* to the contained pages. The timestamp in the second column shows the time of the last reference to the page in the corresponding buffer frame at the time the list (LRUqueue0 or LRUqueue1) was sorted—it is not updated after the creation of the list.

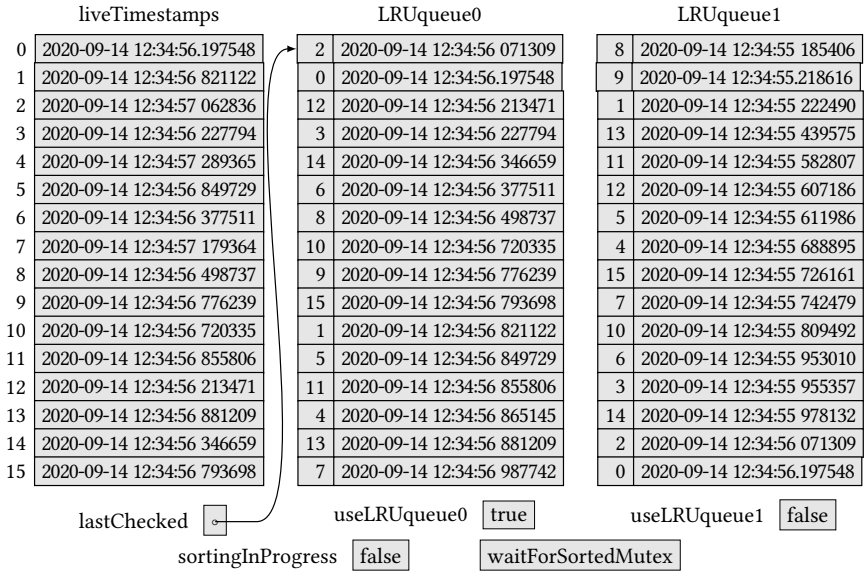


Figure 1.6: The page reference statistics of the Timestamp-Sorting implementation of the LRU page replacement strategy one eviction after sorting LRUqueue0

The *lastChecked* variable refers to the entry in the currently used sorted LRU queue that was *last considered for eviction*. If the currently used queue is *too outdated*, the other queue is prepared by sorting the buffer frames into it based on the current timestamps. During this process, *sortingInProgress* is set and the *waitForSortedMutex* is acquired by each working thread *waiting for this update*.

useLRUqueue0, useLRUqueue1, lastChecked and the liveTimestamps array provide *atomic read and write* operations and sortingInProgress provides *test-and-set* and *clear* operations.

Page Hits The Timestamp-Sorting implementation of the LRU page replacement algorithm prevents the synchronization overhead for page hits by atomically writing the current *wall clock* time to the element in the liveTimestamps array associated with the referenced buffer frame during a page hit. To obtain correctly ordered reference timestamps, a monotonic

clock should be used if available. To further reduce the cost of page eviction, the timestamps are also updated when a page *cannot be evicted temporarily*. Since this indicates that a page is a somewhat hot, it should not be considered for eviction again so quickly. As a small optimization, pages that *cannot be evicted at all* are given a timestamp that is as far in the future as possible (which is always considered the most recent), but a regular page reference to these pages overwrites this timestamp again.

Page Eviction The page in the buffer pool with the *oldest reference timestamp* is selected for eviction whenever a buffer frame needs to be freed. But to avoid an exhaustive search for the oldest reference timestamp in the liveTimestamps array, which would be required each time a buffer frame is needed to be reclaimed, an array of buffer frame indexes *sorted by reference timestamp* is created, in which in the best case the oldest reference timestamp can be found in $\mathcal{O}(1)$. This array does not receive live updates, but is created each time the old version became too outdated. To allow the creation of a new sorted array in the background, two arrays are used alternately, and therefore either LRUqueue0 or LRUqueue1 contains the most up-to-date list of sorted buffer frames. And to ensure that the entry for the buffer frame with the oldest reference timestamp in this list is not outdated, the reference timestamp of the one used to create this list is compared with the live version. A reference to this page made in the meantime (since the last sorting) can be recognized in that way, and due to the fact that this reference was made later than any other page reference used to create the currently used sorted list, this page can be skipped without problems.

Sorting the reference timestamp becomes a *scalability bottleneck* due to its computational complexity of $\mathcal{O}(n \cdot \log(n))$, but asynchronous execution helps with this problem to some degree.

As an example for this algorithm, figure 1.6 shows LRUqueue0 as the most up-to-date list of buffer frames sorted by reference timestamp (useLRUqueue0 is true), and therefore it is iterated through this array when pages are evicted. The last buffer frame (potentially) reclaimed here was buffer frame 2 (lastChecked points to 0 and useLRUqueue0 is true), and therefore its timestamp was updated in liveTimestamps after LRUqueue0 had been sorted—the newly fetched page was referenced or it was found

```
 1: function SELECT
 2:     while true do
 3:         if useLRUqueue0 = true then
 4:             if lastChecked > 0.75 · maxBufferFrameIndex ∧ ¬sortingInProgress.TEST_AND_SET then
 5:                 waitForSortedMutex.ACQUIRE
 6:                 SORTINTO(LRUqueue1)
 7:                 useLRUqueue1 ⟵ true
 8:                 lastChecked ⟵ −1
 9:                 waitForSortedMutex.RELEASE
10:                 useLRUqueue0 ⟵ false
11:                 sortingInProgress.CLEAR
12:             else
13:                 checkThis ⟵ ++lastChecked
14:                 if checkThis > maxBufferFrameIndex then
15:                     waitForSortedMutex.ACQUIRE
16:                     waitForSortedMutex.RELEASE
17:                 else
18:                     if LRUqueue0[checkThis].timestamp = liveTimestamps[LRUqueue0[checkThis].bufferIndex] then
19:                         return LRUqueue0[checkThis].bufferIndex
20:                     else
21:                         continue
22:                     end if
23:                 end if
24:             end if
25:         else if useLRUqueue1 = true then
26:             ...                                   ▷ As before but with LRUqueue1 instead of LRUqueue0
27:         else if ¬sortingInProgress.TEST_AND_SET then
28:                                                   ▷ Initially neither useLRUqueue0 nor useLRUqueue1 is true.
29:             waitForSortedMutex.ACQUIRE
30:             SORTINTO(LRUqueue0)
31:             useLRUqueue0 ⟵ true
32:             lastChecked ⟵ −1
33:             waitForSortedMutex.RELEASE
34:             useLRUqueue1 ⟵ false
35:             sortingInProgress.CLEAR
36:         else
37:             waitForSortedMutex.ACQUIRE
38:             waitForSortedMutex.RELEASE
39:         end if
40:     end while
41: end function
```

Algorithm 1.5: Selection of eviction candidates by the Timestamp-Sorting implementation of the LRU page replacement strategy

temporarily unreclaimable. The next reclaimed buffer frame will be buffer frame 0. It is already known that buffer frames 4 and 7 will be skipped because the contained pages were referenced after LRUqueue0 had been sorted—their timestamps in LRUqueue0 are not up-to-date.

If a buffer frame is to be reclaimed by a working thread and lastChecked is 12 (75 % of buffer frames have been checked out), that thread sets sortingInProgress, acquires the waitForSortedMutex, and sorts the buffer frames in LRUqueue1 based on the timestamps in liveTimestamps as shown in Algorithm 1.5. Other working threads reclaiming buffer frames still use LRUqueue0 until the working thread currently sorting sets useLRUqueue1 to true. They know that they do not need to perform the sorting because they find sortingInProgress set. If all buffer frames are checked out by the other working threads before LRUqueue1 is completely sorted, threads that need to reclaim buffer frames wait using waitForSortedMutex.

1.2.5 Most Recently Used (MRU)

MRU is for LRU, what FILO is for FIFO. With the MRU page replacement strategy, the *most recently referenced* is always evicted. In an OLTP system, this leads to the frequent eviction of hot pages, as these are very often the most recently referenced pages. But due to the arbitrariness of the transaction mix concurrently running on OLTP systems, this eviction strategy will result in rather random eviction decisions.

Like the FILO page replacement strategy, this one has the problem that pages, which are referenced early on after the system cold start, tend to stay in the buffer pool for a very long time, and that they are never evicted if they are never referenced again after the first page eviction (of any page) from the buffer pool. There is no use case in an OLTP system where this page replacement strategy is better suited than every other.

1.2.5.1 Quasi-MRU

Introduction The Quasi-MRU implementation of this page replacement strategy is based on the Quasi-FIFO implementation of the FIFO page replacement strategy but with more sophisticated updates to the page reference statistics.

Page Reference Statistics The Quasi-MRU page replacement algorithm uses—like FILO—a stack (MRU stack) and a queue (retry queue) for its page reference statistics. But both data structures are implemented using a

doubly-linked list in a hash map (shown in figure 1.5) like the one used for the Hash-Map-Doubly-Linked-List implementation of the LRU page replacement strategy. In addition to the operations for enqueuing (alias "push to back") (Algorithm 1.2), dequeuing (alias "pop from front") (Algorithm 1.3) and removing (Algorithm 1.4) a specific buffer pool index, an operation "push to front" (Algorithm 1.6) needs to be provided by the doubly-linked list in a hash map.

```
 1: function pushToFront(key)
 2:     if |hashMap| ≠ 0 then
 3:         if hashMap [key] ≠ NULL then
 4:             return ERROR!
 5:         end if
 6:         hashMap [key] ← (NULL, head)
 7:         hashMap [head] .previous ← key
 8:         head ← key
 9:     else
10:         hashMap [key] ← (NULL, NULL)
11:         tail ← key
12:         head ← key
13:     end if
14: end function
```

Algorithm 1.6: Push an index to the front of Hash-Map-Doubly-Linked-List

Per working thread, the variables currentListChecks and currentlyCheckingRetryQueue are used to balance the page evictions between the MRU stack and the retry queue, as done in the Quasi-FIFO page replacement algorithm.

Updates of the Page Reference Statistics The page reference statistics are not updated on *page hit* because moving a buffer frame to the front of the MRU stack would make that buffer frame the last referenced one and therefore the contained page would probably be selected for eviction while it is still fixed. The update is done when the page gets *unfixed*.

A buffer frame that is *temporarily or permanently unreclaimable* is moved to the end of the retry queue, and an *explicitly reclaimed* buffer frame is removed either from the MRU stack or from the retry queue—wherever it was before. *Empty buffer frames* are always neither in the MRU stack nor in the retry queue.

Page Eviction The selection of buffer frames for page eviction works similar to that of the Quasi-FIFO page replacement algorithm presented in Algorithm 1.1. However, instead of dequeuing buffer frame indexes from the head of an LRU queue, they are popped from the top (tail) of the MRU stack. A notExplicitlyEvictedList does not need to be checked because empty buffer frames are simply not in the MRU stack or in the retry queue. But unlike the simpler lock-free ring buffer used in the Quasi-FIFO page replacement algorithm, the double-linked-lists in hash maps used here require a global latch to be acquired for synchronization.

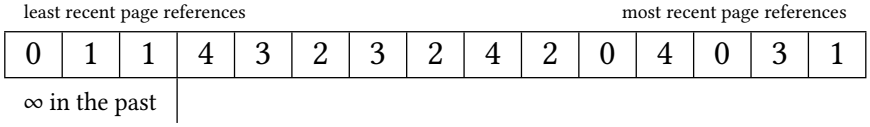
1.2.6 LRU-K

The LRU-K page replacement strategy is an optimization (and generalization) of the LRU page replacement strategy proposed by O'Neil, O'Neil and Weikum in [OOW93]. By taking into account the K *most recent references* to each page in the buffer pool, the authors achieve a page replacement policy that can better discriminate between pages that are referenced *frequently* and those that are referenced *only (once or) a few times*. The fact that the K^{th} most recent reference of a page referenced $<K$ times is considered to be *infinitely far in the past* leads to scan resistance thus overcoming one of the major weaknesses of LRU. *LRU-1* is equivalent to LRU.

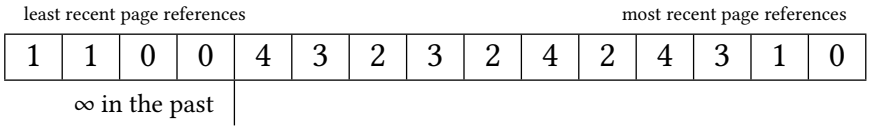
The LRU-K page replacement algorithm evicts the page from the buffer pool where the K^{th} most recent reference is farthest in the past. Therefore, no page with $\geq K$ references will be evicted if pages with $<K$ references are in the buffer pool. *A different eviction policy* is required for the pages with $<K$ references because the K^{th} most recent reference of all these pages is considered to be infinitely far in the past, but a certain order must be determined to evict these pages. One option would be to use LRU for these pages, taking into account only the most recent page reference to these pages. Another possibility—which was used for this evaluation—is to use FIFO to differentiate between pages with $<K$ references. But for $K > 2$ the consideration of the reference frequency of the pages would be a possible solution as well.

$(K - 1)$ *pseudo-references*—which are considered to be infinitely far in the past—are added to the page reference statistics when a page is fetched into the buffer pool. These are considered to be later than any other pseudo-

→ For p_i, i.e. the page that is currently in the buffer frame i, the page reference string $p_4p_3p_2p_3p_2p_4p_2p_0p_4p_0p_3p_1$ (pages replaced in the meantime ignored) leads to the following page reference statistics:



→ After the replacement of p_0 with $p_{0'}$:



→ After a reference to p_1:

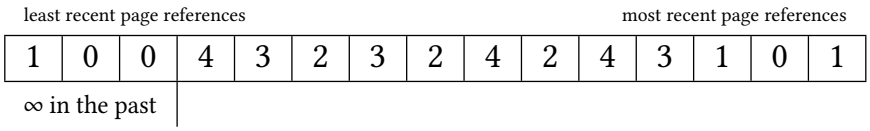


Figure 1.7: Page reference statistics of LRU-3 for 5 buffer frames

references currently contained—and are therefore inserted after them. This leads to the eviction of the *oldest page with <K page references* (FIFO) when the most recent page reference from the page reference statistics is selected for eviction.

Figure 1.7 shows the abstract page reference statistics for a given page reference string collected by the LRU-3 page replacement policy with pseudo-references inserted as described above. The page in buffer frame 0 was referenced twice (1 pseudo-reference) and the page in buffer frame 1 was referenced just once (2 pseudo-references). The page p_0 was *replaced* by the page $p_{0'}$, because it was *fetched into the buffer pool before* page p_1 (FIFO). The references to p_0 have been removed and 2 $(K - 1)$ pseudo-references have been inserted after the most recent pseudo-reference in the page reference statistics (that of p_1). The least recent reference to the p_1 page (including pseudo-references) is removed when a new reference to this page is recorded.

1.2.6.1 Hash-Map-Doubly-Linked-List Implementation

Introduction The Hash-Map-Doubly-Linked-List implementation of the LRU-K page replacement policy is based on the same principle as the Hash-Map-Doubly-Linked-List Implementation of LRU. But for each buffer frame it manages the K most recent references in the used LRU queue.

Page Reference Statistics To *identify the K references recorded per buffer frame* in the LRU queue, a unique ID must be defined for these references. The K most recent references to the page in buffer frame i receive IDs in the range $i \cdot K$ to $i \cdot K + (K - 1)$.

When a page p_i (in buffer frame i) is re-referenced, the K^{th} least recent reference to p_i is removed from the LRU queue and enqueued in the LRU queue as the most recent reference. However, for these operations the ID ID_{i_k} of the K^{th} least recent reference to p_i is required. It can be found by traversing the LRU queue from the head (least recent references) to the tail until a reference ID ID_j with $ID_j/K = i$ is found. Due to the fact that this traversal is done in $\mathcal{O}(n)$ (for very frequently referenced pages, almost $K \cdot n$ elements in the LRU queue must be visited), maintaining the ID of the K^{th} least recent reference to p_i (in frameReference$[i]$) is a *significant optimization* of the function used to update page reference statistics on page hit.

To allow the insertion of $(K - 1)$ pseudo-references—between the most recent pseudo-reference and the least recent real reference in the LRU queue—when a page is fetched into the buffer pool, the ID of the least recent real reference is maintained in the variable leastRecentlyUsedFinite.

The page reference statistics are protected against data races caused by concurrent page references by the LRUqueueLatch.

Page Hit Whenever a page hit (or alternatively a page unfix) occurs, the least recent reference to that page is removed from the LRU queue (LRUqueue) and enqueued as the most recent reference in the LRU queue. Algorithm 1.7 shows the corresponding function. The used functions of the Hash-Map-Doubly-Linked-List are described in Algorithms 1.8, 1.4 and 1.2.

```
1: function onHit(bufferIndex)
2:     LRUqueueLatch.acquire
3:     referenceID ← (frameReference[bufferIndex] mod K) + (K · bufferIndex)
4:     if referenceID = leastRecentlyUsedFinite then
5:         if LRUqueue.getAfter(leastRecentlyUsedFinite) = NULL then
6:             leastRecentlyUsedFinite ← NULL
7:         else
8:             leastRecentlyUsedFinite ← LRUqueue.getAfter(leastRecentlyUsedFinite)
9:         end if
10:    end if
11:    LRUqueue.remove(referenceID)
12:    if leastRecentlyUsedFinite = NULL then
13:        leastRecentlyUsedFinite ← referenceID
14:    end if
15:    LRUqueue.enqueue(referenceID)
16:    frameReference[bufferIndex] ++
17:    LRUqueueLatch.release
18: end function
```

Algorithm 1.7: Update the page reference statistics on a page hit when the Hash-Map-Doubly-Linked-List implementation of the LRU-K page replacement policy is used

```
1: function getAfter(key)
2:     if hashMap[key] = NULL then
3:         return ERROR!
4:     else if tail = key then
5:         return NULL
6:     else
7:         return hashMap[key].next
8:     end if
9: end function
```

Algorithm 1.8: Get the index after a given index in a Hash-Map-Doubly-Linked-List

Page Eviction An eviction candidate is selected by dequeuing the least recent reference (divided by K) from the head of the LRU queue. Algorithm 1.9 shows the corresponding function. The used functions of the Hash-Map-Doubly-Linked-List are described in Algorithms 1.10, 1.8 and 1.4.

If the selected eviction candidate cannot be evicted, the reference ID dequeued from the LRU queue is enqueued (algorithm 1.2) again. Algorithm 1.11 describes the corresponding function.

But if a new page can be fetched into the reclaimed buffer frame, **function** insert described in Algorithm alg:hashmapdoublylinkedlistlrukinsert is called. It begins with resetting the frameReference variable, then it removes

```
 1: function SELECT
 2:     LRUqueueLatch.ACQUIRE
 3:     front ← LRUqueueLatch.GETFRONT
 4:     if referenceID = leastRecentlyUsedFinite then
 5:         if LRUqueue.GETAFTER(leastRecentlyUsedFinite) = NULL then
 6:             leastRecentlyUsedFinite ← NULL
 7:         else
 8:             leastRecentlyUsedFinite ← LRUqueue.GETAFTER(leastRecentlyUsedFinite)
 9:         end if
10:     end if
11:     LRUqueue.REMOVE(referenceID)
12:     return front/K
13: end function
```

Algorithm 1.9: Selection of eviction candidates by the Hash-Map-Doubly-Linked-List implementation of the LRU-K page replacement strategy

```
1: function GETFRONT
2:     if |hashMap| = 0 then
3:         return ERROR!
4:     else
5:         return head
6:     end if
7: end function
```

Algorithm 1.10: Get head index of an Hash-Map-Doubly-Linked-List

all the references associated with the buffer frame. It then inserts the $(K - 1)$ pseudo-references into the LRU queue, and finally enqueues a reference for the initial reference to the buffer frame. The same function is also used when a buffer frame is used for the first time, so the LRUqueueLatch must be acquired first and released twice after the function—once for acquisition in this function and once for potential acquisition in the previously called **function** SELECT. The Hash-Map-Doubly-Linked-List functions used are described in the algorithms 1.8, 1.4, 1.13 and 1.2.

1.2.6.2 Timestamp-Sorting Implementation

Introduction The Timestamp-Sorting implementation of the LRU-K page replacement policy is based on the same principle as the Timestamp-Sorting Implementation of LRU. Therefore, the synchronization overhead for page hits is eliminated, resulting in better scalability. But for each buffer frame it manages the timestamps of the K most recent references.

```
1: function notEvictabl
2:     referenceID ← (frameReference[bufferIndex]   mod K) + (K · bufferIndex)
3:     if leastRecentlyUsedFinite = NULL then
4:         leastRecentlyUsedFinite ← referenceID
5:     end if
6:     LRUqueue.enqueue(referenceID)
7:     frameReference[bufferIndex] ++
8:     LRUqueueLatch.release
9: end function
```

Algorithm 1.11: Update the page reference statistics for an eviction candidate that cannot be evicted when the Hash-Map-Doubly-Linked-List implementation of the LRU-K page replacement policy is used

Page Reference Statistics The page reference statistics of the Timestamp-Sorting implementation of LRU-K is very similar to that of LRU. But in this case, liveTimestamps is not a *two-dimensional* array with K *columns* (timestamps) per buffer frame. The array liveTimestampsOldestTimestamp contains for each buffer frame the array index of the least recent of the K timestamps in the array liveTimestamps. This value is used whenever the reference statistic of that particular buffer frame is updated or when the least recent timestamps of each buffer frame are sorted into LRUqueue0 or LRUqueue1. The pseudo-references use a timestamp as far in the past as possible, but to allow FIFO page eviction to be used for pages with $<K$ page references, the timestamps used for pseudo-references are incremented using the infinitePast counter.

Updates of the Page Reference Statistics On a *page hit* (or alternatively a page unfix), the least recent timestamp of that page (according to liveTimestampsOldestTimestamp) is overwritten with the current time and the value in liveTimestampsOldestTimestamp is atomically incremented. This also happens when a page *temporarily cannot be evicted*.

K timestamps that are as far in the future as possible are used for pages that cannot be evicted at all. However, if such a page is referenced, the "least recent" timestamp will be overwritten with the current time.

As already described, the timestamps used for the pseudo-references are taken from the infinitePast counter. This means, when a page is fetched into the buffer pool, $(K - 1)$ timestamps are set to infinitePast which is incremented after each read and one timestamp is set to the current time.

```
 1: function INSERT(bufferIndex)
 2:     LRUqueueLatch.ACQUIRE
 3:     frameReference [bufferIndex] ← 0
 4:     for i ← 0 to K − 1 do
 5:         referenceID ← (i  mod K) + (K · bufferIndex)
 6:         if referenceID = leastRecentlyUsedFinite then
 7:             if LRUqueue.GETAFTER(leastRecentlyUsedFinite) = NULL then
 8:                 leastRecentlyUsedFinite ← NULL
 9:             else
10:                 leastRecentlyUsedFinite ← LRUqueue.GETAFTER(leastRecentlyUsedFinite)
11:             end if
12:         end if
13:         LRUqueue.REMOVE(referenceID)
14:     end for
15:     referenceID ← (frameReference [bufferIndex]  mod K) + (K · bufferIndex)
16:     if leastRecentlyUsedFinite = NULL then
17:         leastRecentlyUsedFinite ← referenceID
18:     end if
19:     if K ≥ 2 then
20:         referenceID ← (frameReference [bufferIndex]  mod K) + (K · bufferIndex)
21:         LRUqueue.INSERTBEFORE(referenceID, leastRecentlyUsedFinite)
22:     end if
23:     for i ← 2 to K − 1 do
24:         beforeID ← (frameReference [bufferIndex]  mod K) + (K · bufferIndex)
25:         frameReference [bufferIndex] ++
26:         referenceID ← (frameReference [bufferIndex]  mod K) + (K · bufferIndex)
27:         LRUqueue.INSERTBEFORE(referenceID, beforeID)
28:     end for
29:     if K ≥ 2 then
30:         frameReference [bufferIndex] ++
31:     end if
32:     LRUqueue.ENQUEUE(referenceID)
33:     frameReference [bufferIndex] ++
34:     LRUqueueLatch.RELEASE
35:     LRUqueueLatch.RELEASE
36: end function
```

Algorithm 1.12: Update the page reference statistics for a fetched page when the Hash-Map-Doubly-Linked-List implementation of the LRU-K page replacement policy is used

Page Eviction The selection of eviction candidates works almost exactly the way as in the Timestamp-Sorting implementation of the LRU page replacement policy. This was described in algorithm 1.5. But both the sorting and the checking whether a page has been referenced in the meantime take into account the liveTimestampsOldestTimestamp array here.

```
 1: function insertBefore(key, ref)
 2:     if hashMap [ref] = NULL then
 3:         return ERROR!
 4:     else if hashMap [key] = NULL then
 5:         if head = ref then
 6:             hashMap [key] ← (NULL, ref)
 7:             head ← key
 8:             hashMap [ref] .previous ← key
 9:         else
10:             hashMap [key] ← (hashMap [ref] .previous, ref)
11:             hashMap [hashMap [ref] .previous] .next ← key
12:             hashMap [ref] .previous ← key
13:         end if
14:     end if
15: end function
```

Algorithm 1.13: Insert an index before another index in an Hash-Map-Doubly-Linked-List

1.2.7 Segmented LRU (SLRU)

Introduction *SLRU*, like LRU-K, is an optimization of the LRU page replacement policy that combines the use of reference recency with reference frequency for eviction decisions. It was proposed by Karedla, Love and Wherry in [KLW94].

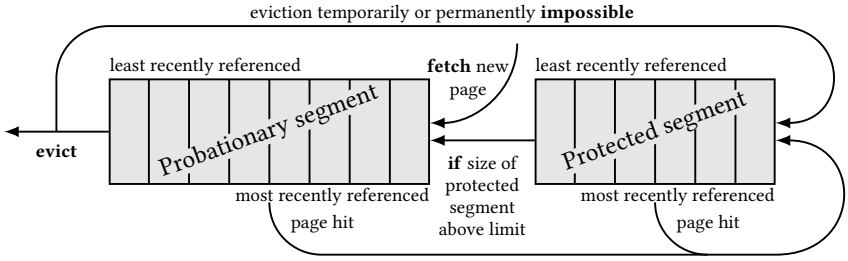


Figure 1.8: The probationary and protected segment of SLRU and the flow of pages between those

Page Reference Statistics The page reference statistics consist of two LRU queues, the probationary segment and the protected segment. The *protected segment* has an adjustable maximum size (usually smaller than the buffer pool), is reserved for pages that are referenced multiple times and is ordered by reference recency. The *probationary segment* contains all

buffered pages that were referenced only once, as well as pages that were referenced multiple times, where the most recent reference is less recent than that of any page in the protected segment.

Updates of the Page Reference Statistics Figure 1.8 illustrates the *flow of pages* in the page reference statistics of the SLRU page replacement policy. Pages that are fetched into the buffer pool are enqueued in the probationary segment. When a page is re-referenced (but not on unfix), it is removed from the segment it is currently in and enqueued in the protected segment—if the number of pages in the protected segment is exceeded, the least recently referenced page in it is dequeued and enqueued in the probationary segment.

Page Eviction The least recently referenced page from the *probationary segment* will be selected as eviction candidate. If it *cannot be evicted*—either temporarily or permanently—it is assumed that it is a hot page and will therefore be enqueued in the protected segment.

1.2.8 CLOCK

Introduction The CLOCK page replacement algorithm is a scalable approximation of the LRU page replacement policy which was first described by Corbató in [Cor69][5]. The implementation of this page replacement algorithm extends the LOOP variant of the FIFO page replacement policy by a so-called *usage-bit*.

Page Reference Statistics The page reference statistics of the CLOCK page replacement algorithm consists of a *global cyclic counter* (representing the clock hand) and an array of *usage-bits*—one for each buffer frame. The usage-bit of a buffer frame is true if the page contained in it has been referenced since the last sweep of the hand over that buffer frame.

The cyclic counter used is the *Modulo Counter*, which was evaluated in my Project Thesis [Gil20]. It uses the modulo (to get a value in the buffer

[5] F. J. Corbató proposed a more general algorithm with some similarities to GCLOCK. If the parameter k of his generalized algorithm is set to 1, the result is the CLOCK page replacement algorithm.

frame index range) of a global counter variable that is incremented using the atomic fetch-and-increment instruction.

Figure 1.9 shows the usage-bits of a buffer pool with 16 buffer frames arranged in a circle with the clock hand pointing to buffer frame 4. The next buffer frame that the clock pointer moves to has index 5. Its usage-bit is false because it has not been referenced since it was last swept.

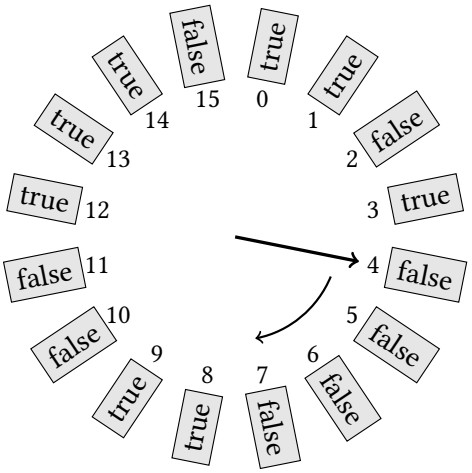


Figure 1.9: The page reference statistics of the CLOCK page replacement algorithm

Updates of the Page Reference Statistics Each time a *page hit* occurs, the usage-bit of the corresponding buffer frame is *set to true* atomically. This can also be done whenever a *page is unfixed* or when a page selected as a candidate for eviction by the CLOCK page replacement algorithm *cannot be evicted*. When a page is fetched into the buffer pool, its corresponding usage-bit can either be set to false, which results in a faster eviction of pages that are *referenced only once*, or it can be set to true. If the use bit is always set to true when the corresponding page is unfixed, an initially false use bit will always be true as soon as the page can be evicted.

Page Eviction The CLOCK page eviction algorithm *moves its clock hand forward*—setting the usage-bits of the buffer frames which have been swept

to true—until it points to a buffer frame where the *usage-bit is false*. This buffer frame is then the selected eviction candidate.

It approximates the LRU page replacement policy by storing—instead of the exact time of the most recent reference of each page—-the information *whether a page was referenced in a particular time frame*—which corresponds to a complete sweep of the clock—in its page reference statistics. With this information, it is not possible to determine the *exact order of the pages* based on the most recent references, but it is guaranteed that the most recent reference of the least recently referenced page was less than the time of a complete sweep of the clock hand before the most recent reference of the page selected as an eviction candidate.

Performance Evaluation Figure 1.10 shows transaction throughput and figure 1.11 shows the miss rate achieved with different variants of the CLOCK page replacement algorithm for different buffer pool sizes. Details of the benchmark setup can be found in section 1.3.

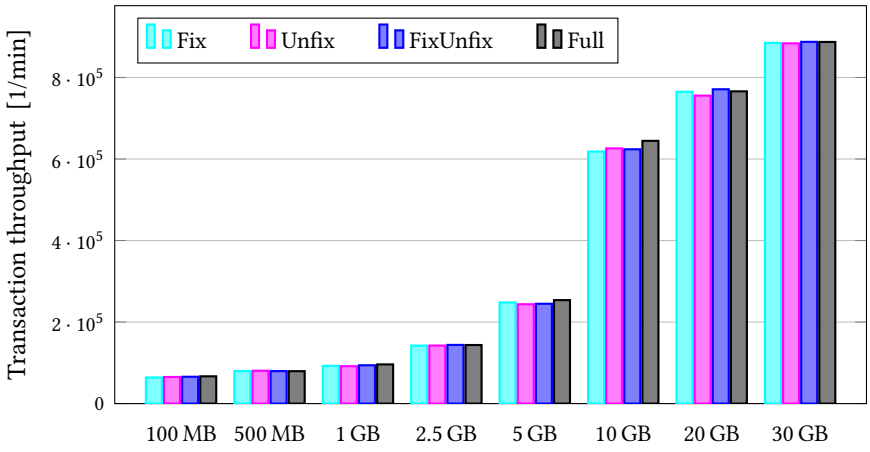


Figure 1.10: Transaction throughput of the CLOCK page replacement algorithm variants for the *TPC-C* benchmark on 100 warehouses

The *Fix* variant sets the corresponding usage-bit to true whenever a page hit occurs. The *Unfix* variant does this on page unfix, and the *FixUnfix* variant combines the two. The *Full* variant does it also if the buffer frame

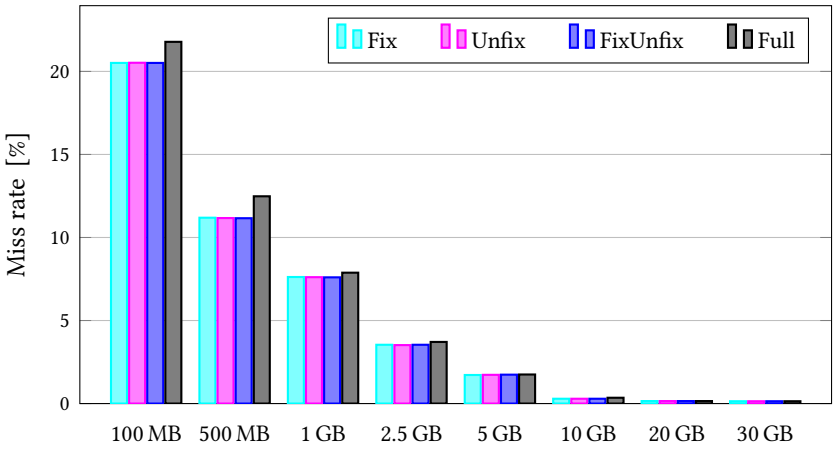


Figure 1.11: Miss rate of the CLOCK page replacement algorithm variants for the *TPC-C* benchmark on 100 warehouses

cannot be reclaimed after it being selected as eviction candidate. Only the *Fix* variant allows the immediate eviction of a page referenced once.

The different variants show only slight differences in performance. The additional overhead for setting the usage-bit to true for the *FixUnfix* variant is negligible and the difference in the resulting miss rates is statistically insignificant. The only difference that can be observed at all is the slightly higher miss rate achieved with the *Full* variant, indicating that a page that cannot be evicted temporarily should not be considered hot. But even if the *Full* variant has the highest overhead, it does not perform worse than the other variants when it comes to the achieved transaction throughput.

The *Full* variant is used for the performance evaluation in section 1.3.

1.2.9 Zero-Handed CLOCK (ZCLOCK)

Introduction The ZCLOCK page replacement algorithm is a variation of the CLOCK page replacement algorithm, in which the *clock hand points to a random buffer frame at every move.* The result is a RANDOM page replacement algorithm in which each buffer frame is given a *second chance* to be referenced again. The random selection is done using *SplitMix32*.

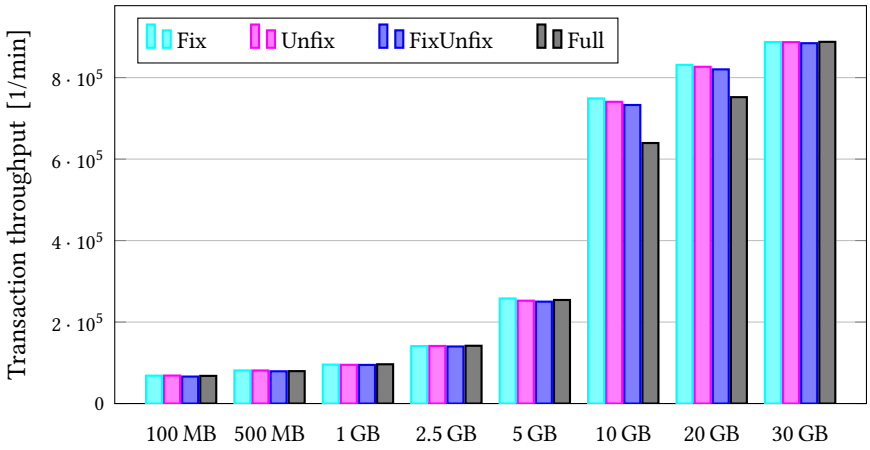


Figure 1.12: Transaction throughput of the ZCLOCK page replacement algorithm variants for the *TPC-C* benchmark on 100 warehouses

Page Reference Statistics The page reference statistics of the ZCLOCK page replacement algorithm for a buffer pool with 16 buffer frame is shown in figure 1.14. There is just one *usage-bit* (but actually no clock hand) for each buffer frame—if true, the corresponding page has been referenced since the clock hand last pointed to it. ZCLOCK uses the usage-bits in the same way as the CLOCK page replacement algorithm. But the random movement of the clock hand makes the *duration between two times the clock hand points to a particular buffer frame completely unpredictable*, resulting in behavior less similar to the LRU page replacement policy.

Performance Evaluation Figure 1.12 shows transaction throughput and figure 1.13 shows the miss rates achieved with different variants of the ZCLOCK page replacement algorithm for different buffer pool sizes. Details of the benchmark setup can be found in section 1.3.

The evaluated variants of ZCLOCK are the same as for CLOCK.

The lower overhead variants of the ZCLOCK page replacement algorithm, *Fix* and *Unfix*, perform best, regardless of buffer pool size. The variant with the highest overhead—*Full*—achieves a lower *transaction throughput*—especially when the database almost fits into the buffer pool.

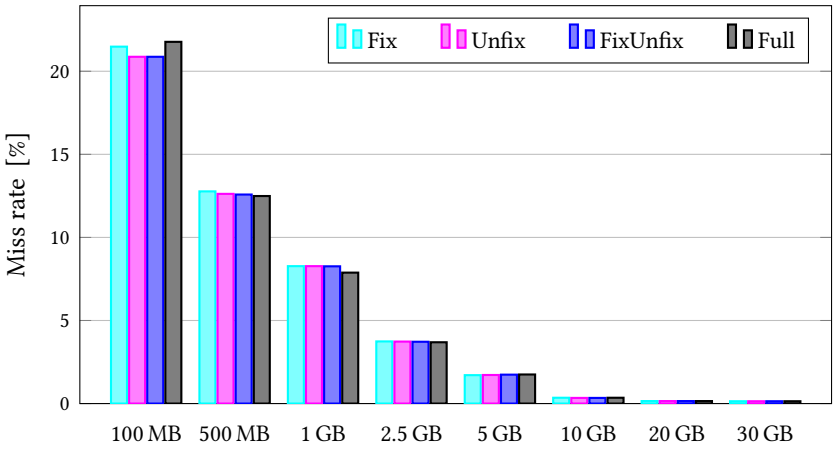


Figure 1.13: Miss rate of the ZCLOCK page replacement algorithm variants for the *TPC-C* benchmark on 100 warehouses

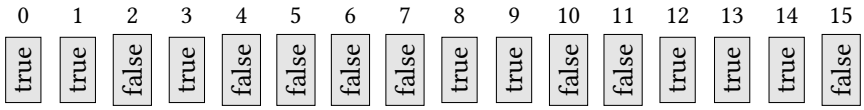


Figure 1.14: The page reference statistics of the ZCLOCK page replacement algorithm

With *larger buffer pools* (≥2.5 GB), the *miss rates* achieved are almost identical for all variants. The small deviations for *smaller buffer pools* are statistically insignificant.

In contrast to the results for the CLOCK page replacement algorithm, the ZCLOCK page replacement algorithm cannot benefit from more information collected in the page reference statistics of the *Full* variant. The less predictable behavior of ZCLOCK—with more or less *random eviction decisions*—cannot properly utilize this information.

The *Fix* variant is used for the performance evaluation in section 1.3.

1.2.10 Generalized CLOCK (GCLOCK)

Introduction The GCLOCK page replacement algorithm is a scalable approximation to the LRU page replacement policy which was proposed by Effelsberg and Härder in [EH84]. A simpler approach was proposed by Corbató in [Cor69] and extended by Smith in [Smi78] by additional rules for prefetching. It *generalizes the CLOCK page replacement algorithm* by replacing the usage-bit with an *usage-count*, thus including information about reference frequency of a page in the page reference statistics.

Page Reference Statistics The page reference statistics maintained by the GCLOCK page replacement algorithm for an exemplary buffer pool with 16 buffer frames is shown in figure 1.15. Instead of one usage-bit per buffer frame, as used by the CLOCK page replacement algorithm, *usage-counts* (of integer type) are used. The *clock hand* is implemented in the same way as in the CLOCK page replacement algorithm, using the *Modulo Counter* from [Gil20].

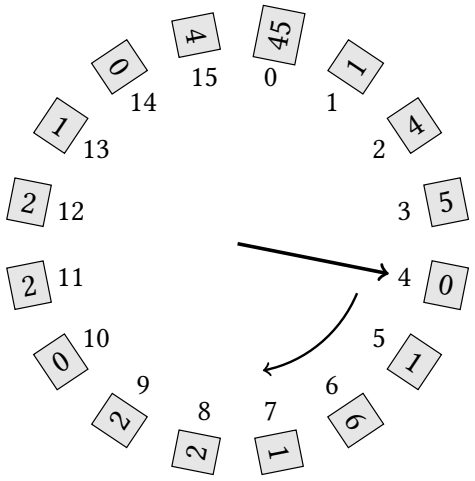


Figure 1.15: The page reference statistics of the GCLOCK and DGCLOCK page replacement algorithms

Updates of Page Reference Statistics Effelsberg and Härder worked out *two different versions* of the GCLOCK page replacement algorithm, GCLOCK-V1 and GCLOCK-V2 with different rules for updating the usage-counts in case of a *page hit* (or alternatively in case of a *page unfix*). The usage-counts can also be updated based on these rules if an *eviction candidate cannot be evicted* temporarily or permanently. The different rules are specified in subsections 1.2.10.1 and 1.2.10.2. Both versions of the GCLOCK page replacement algorithm define a parameter F to which a corresponding usage-count is set ($UC(p) = F$) *when a page p is fetched* into a buffer frame.

For optimal hit rates, the values of all the parameters defined by the GCLOCK page replacement algorithms needs to be chosen specifically for every application. The optimization effort is likely to prevent the usage of these page replacement algorithms in any production DBMS.

Page Eviciton Both versions of the evict pages the same way. They *move the clock hand forward*—decrementing the usage-counts of the buffer frames which have been touched by the value of a new parameter S (which is not part of the definition in [EH84])—until it points to a buffer frame where the *usage-count is* 0. This buffer frame is then the selected eviction candidate.

1.2.10.1 GCLOCK-V1

Introduction The GCLOCK-V1 page replacement algorithm is the version of the GCLOCK page replacement algorithm where *reference recency* and *reference frequency* are taken into account for eviction decisions. One or the other factor can be taken into account to a greater extent by choosing the appropriate parameter values.

Updates of Page Reference Statistics The GCLOCK-V1 page replacement algorithm *increases* the corresponding usage-count ($UC(p) = UC(p) + R$) of a page p by the value of the parameter R on page hit (or alternatively on page unfix). In this way, the usage-count of a page represents the number of references to this page. The decrease of the usage-counts during eviction can be seen as an *aging process*. With a greater value for R, eviction

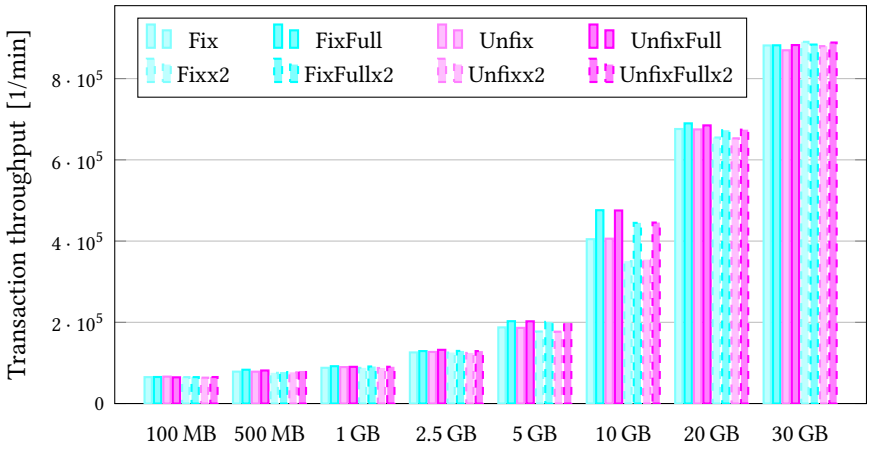


Figure 1.16: Transaction throughput of the GCLOCK-V1 page replacement algorithm variants for the *TPC-C* benchmark on 100 warehouses

decisions are based more on *reference frequency* while a greater S result in an increased focus on *reference recency*.

Performance Evaluation Figure 1.16 shows transaction throughput and figure 1.17 shows the miss rate achieved with different variants of the GCLOCK-V1 page replacement algorithm for different buffer pool sizes. Details of the benchmark setup can be found in section 1.3.

The *Fix* variant uses $F = 25$ when a page is fetched into the buffer pool and $R = 5$ when a page is fixed, while the *FixFull* variant also updates usage numbers based on R when an eviction candidate cannot be evicted. The *Unfix* and *UnfixFull* variants are the equivalents for updates on page unfix. The variants ending on *x2* use $F = 50$ and $R = 10$—which both increases overhead and improves the information available for eviction decisions. The S parameter is always 1, which gives the GCLOCK-V1 variants a focus on the reference frequency.

For the *most buffer pool sizes* the *FixFull* variants outperform the *Fix* variants and the *UnfixFull* variants outperform the *Unfix* variants in terms of transaction throughput. If the buffer pool is *slightly smaller than the initial DB* (10 GB), the performance difference is about 15 %. The difference

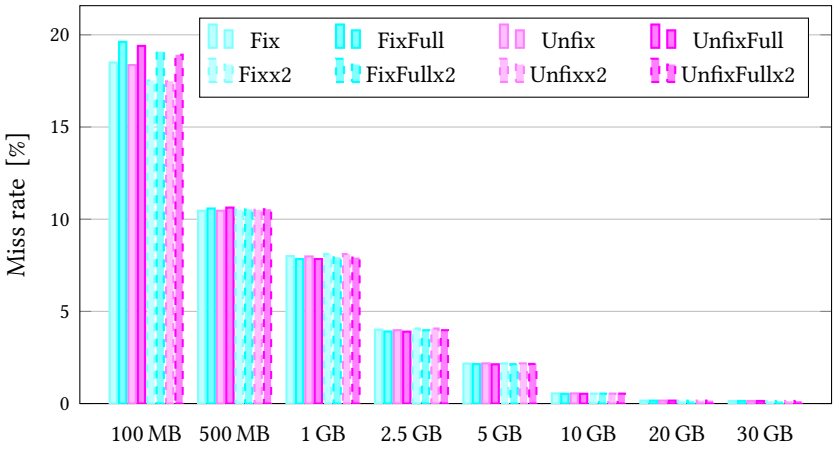


Figure 1.17: Miss rate of the GCLOCK-V1 page replacement algorithm variants for the *TPC-C* benchmark on 100 warehouses

is smaller for *smaller buffer pools*, and for a 100 MB buffer pool, the *Fix* and *Unfix* variants are even slightly faster. Once the buffer pool is *larger than the database*, the performance differences become statistically insignificant (±0.5 %). The performance differences between the *Fix* (*FixFull*) and the *Unfix* (*UnfixFull*) variants are negligible (±2 %).

The *miss rates* of the *FixFull* and *UnfixFull* variants are 0.53 % if the buffer pool size is 10 GB, while they are 0.55 % for the *Fix* and *Unfix* variants. For the *x2* variants, the miss rates are 0.54 %. The *miss rate* of the *FixFull* and *UnfixFull* variants is also lower for a buffer pool of 1 GB–5 GB. The higher miss rates of the *FixFull* and *UnfixFull* variants for the buffer pool size 500 MB are counterintuitive considering the transaction throughput achieved. Once the *complete database fits into the buffer pool*, the miss rates of all variants are identical.

The almost non-existent performance and miss rate difference when the *database is completely in memory* is to be expected, since in this case no pages need to be evicted. Therefore, the only overhead due to the page replacement algorithm is the incrementing of the usage-count on either page fix or page unfix. The negligible difference in performance between the *Fix* (*FixFull*) and the *Unfix* (*unfixFull*) variants is the result of the fact,

that the clock hand passes a page ≥25× between fetching the page into the buffer pool and evicting it from it for *Fix* (*FixFull*), but only once more for *Unfix* (*UnfixFull*). The lower miss rate and better performance of the *FixFull* and *UnfixFull* variants for *medium-sized buffer pools* suggest that the pages that cannot be evicted temporarily will be re-referenced in the near future. The decreasing differences in performance for *smaller buffer pools* are the result of shorter clock hand cycle times, which leads to a smaller effect of this additional increase of the usage-count.

The *FixFull* variant is used for the performance evaluation in section 1.3.

1.2.10.2 GCLOCK-V2

Introduction The GCLOCK-V2 page replacement algorithm is the version of the GCLOCK page replacement algorithm where only *reference recency* is taken into account for eviction decisions. GCLOCK-V2 with $R = 1$ is equivalent to CLOCK.

Updates of Page Reference Statistics The GCLOCK-V2 page replacement algorithm *sets* the corresponding usage-count ($UC(p) = R$) of a page p to the value of the parameter R on page hit (or alternatively on page unfix). Higher R result in more precise information about how many page misses have occurred in the DBMS (which corresponds to the time elapsed) since the most recent reference of a page. This allows a *better approximation* to the LRU page replacement algorithm.

Performance Evaluation Figure 1.18 shows transaction throughput and figure 1.19 shows the miss rate achieved with different variants of the GCLOCK-V2 page replacement algorithm for different buffer pool sizes. Details of the benchmark setup can be found in section 1.3.

The evaluated variants of GCLOCK-V2 and the corresponding parameter values are the same as for GCLOCK-V1.

The differences in transaction throughput between the different variants of the GCLOCK-V2 page replacement algorithm are smaller than the differences between the variants of the GCLOCK-V1 page replacement algorithm. As with GCLOCK-V1, the performance differences are negligible or even statistically insignificant for very small and sufficiently large buffer

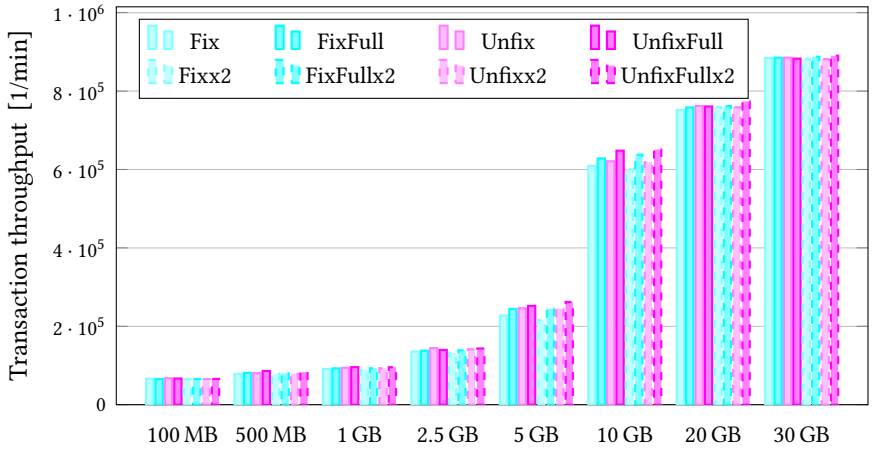


Figure 1.18: Transaction throughput of the GCLOCK-V2 page replacement algorithm variants for the *TPC-C* benchmark on 100 warehouses

pools. However, it is an interesting detail that the *Unfix* (*UnfixFull*) variants perform slightly better than the *Fix* (*FixFull*) variants.

It is absolutely counterintuitive that the miss rates of the *Full* variants are greater than that of the corresponding other variants. But the *Unfix* (*UnfixFull*) variants have slightly lower miss rates than the *Fix* (*FixFull*) variants, which explains the higher transaction throughput.

The higher performance of the *Unfix* and *UnfixFull* variants suggests that the chosen value for the F parameter is too high. If a page that is referenced only once is unfixed, the *Unfix* and *UnfixFull* variants set the usage-count to R, which is one-fifth of F. The *Fix* and *FixFull* variants do not do this for these pages, so they stay in the buffer pool much longer without being referenced again. The less superior results for the *Full* variants—compared to these variants with GCLOCK-V1—suggest that pages that could not be evicted temporarily are subsequently referenced more frequently than others, which is taken into account by GCLOCK-V1 but not by the GCLOCK-V2 page replacement algorithm.

The *UnfixFull* variant is thus used for the performance evaluation in section 1.3.

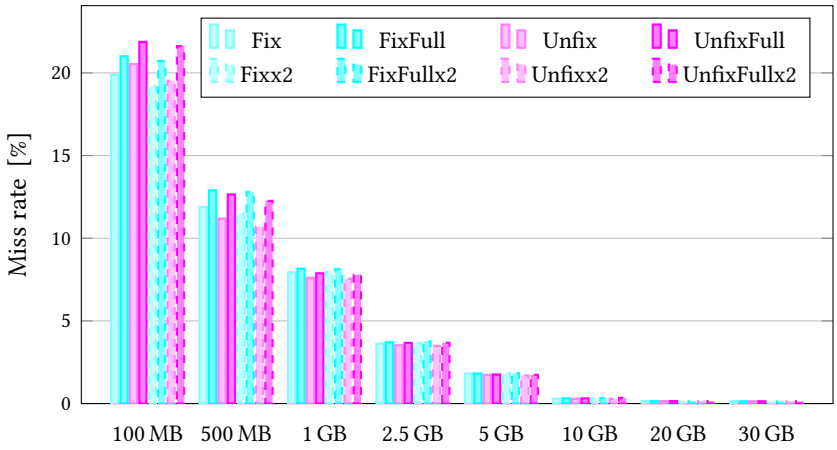


Figure 1.19: Miss rate of the GCLOCK-V2 page replacement algorithm variants for the *TPC-C* benchmark on 100 warehouses

1.2.11 Dynamic Generalized CLOCK (DGCLOCK)

Introduction The DGCLOCK page replacement algorithm is a variant of the GCLOCK page replacement algorithm, which was also proposed by Effelsberg and Härder in [EH84], where changes to the usage-count of a buffer frame depend on the *type of page* it contains.

Page Reference Statistics The page reference statistics of the DGCLOCK page replacement algorithm are *identical* to those of the GCLOCK page replacement algorithm. Examplary page reference statistics are given in figure 1.15.

Updates of Page Reference Statistics *Both versions* of the GCLOCK page replacement algorithm—GCLOCK-V1 and GCLOCK-V2—have *"dynamic" equivalents* under DGCLOCK. DGCLOCK-V1 and DGCLOCK-V2 define different rules for the updating of usage-counts of buffer frames on corresponding *page hits* (or alternatively on *page unfixes*), and optionally these rules are also applied if a *page cannot be evicted* temporarily or permanently. The rules used by the two versions of DGCLOCK are specified in

the sections 1.2.11.1 and 1.2.11.2. The usage-count associated with a *newly fetched page* is set to the value of parameter F_i for a page of type i.

Since DGCLOCK has even more parameter values to select than GCLOCK, optimization for specific applications is even more challenging. In a relational DBMS, values for the parameters F_i and R_i must be selected for a variety of page types i:

- regular data pages
- LOB data pages
- B-tree index pages of different levels and of primary and secondary indexes
- hash index pages of primary and secondary indexes
- pages of other index implementations (bitmap, R-tree etc.)
- dirty pages
- other metadata pages

Page Eviction The DGCLOCK page replacement algorithms select the eviction candidates in *exactly the same way* as the GCLOCK page replacement algorithms—the implementation is absolutely identical.

1.2.11.1 DGCLOCK-V1

Introduction The only difference between the DGCLOCK-V1 page replacement algorithm and its "static" counterpart GCLOCK-V1 is that instead of one parameter R for all pages, there are many *parameters R_i for different page types i*. A larger value R_i for a specific page type allows prioritizing this page type i in the buffer pool.

Updates of Page Reference Statistics The DGCLOCK-V1 page replacement algorithm *increases* the corresponding usage-count ($UC(p) = UC(p) + R_p$) of a page p by the value of the parameter R_p, which is specific to the page type of p, on page hit (or alternatively on page unfix). Page types i with greater values of R_i are prioritized and therefore corresponding pages are evicted later.

Performance Evaluation Figure 1.20 shows transaction throughput and figure 1.21 shows the miss rate achieved with different variants of the

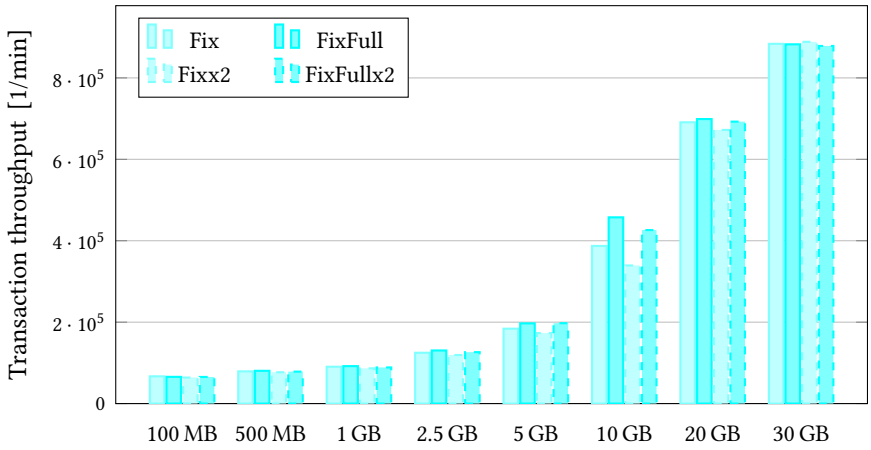


Figure 1.20: Transaction throughput of the DGCLOCK-V1 page replacement algorithm variants for the *TPC-C* benchmark on 100 warehouses

DGCLOCK-V1 page replacement algorithm for different buffer pool sizes. Details of the benchmark setup can be found in section 1.3.

The *Fix* variant uses $F_{\text{non-B-tree}} = 25$, $F_{\text{root}} = 25$, $F_{\text{root-1}} = 10$ and $F_{\text{root-}\geq 2} = 5$ when a page is fetched into the buffer pool and $R_{\text{non-B-tree}} = 5$, $R_{\text{root}} = 5$, $R_{\text{root-1}} = 2$ and $R_{\text{root-}\geq 2} = 1$ when a page is fixed, while the *FixFull* variant also updates usage numbers based on R when an eviction candidate cannot be evicted. The variants ending on *x2* use $F_{\text{non-B-tree}} = 50$, $F_{\text{root}} = 50$, $F_{\text{root-1}} = 25$, $F_{\text{root-}\geq 2} = 10$, $R_{\text{non-B-tree}} = 10$, $R_{\text{root}} = 10$, $R_{\text{root-1}} = 5$ and $R_{\text{root-}\geq 2} = 2$—which both increases overhead and improves the information available for eviction decisions. The S parameter is always 1, which gives the DGCLOCK-V1 variants a focus on the reference frequency.

For buffer pool sizes ≥500 MB, the DGCLOCK-V1 variants with the higher value of F_i and R_i (which end on *x2*) show a slightly worse hit rate and transaction throughput than the other variants. The *FixFull* variant performs best for all these buffer pool sizes. For the 100 MB buffer pool size, the *Fixxx2* variant achieves the lowest miss rate, but the *Fix* variant still achieves the higher transaction throughput.

The *x2* variants better approximate the behavior of the LRU page replacement policy, but the minimally higher miss rates of these variants for buffer

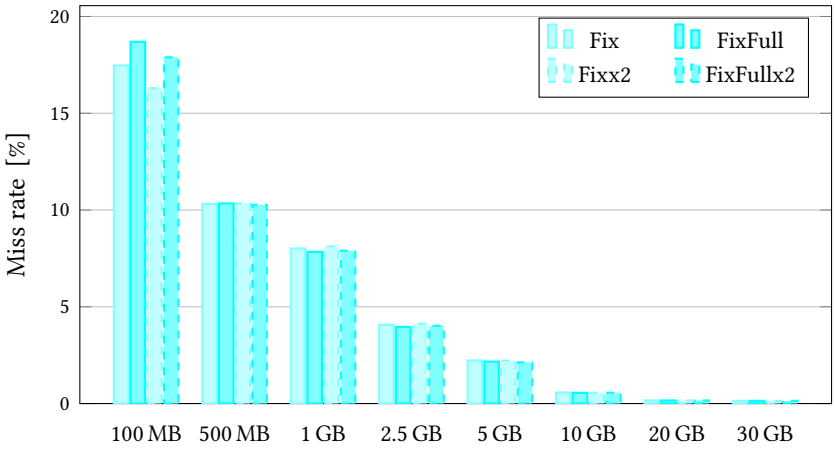


Figure 1.21: Miss rate of the DGCLOCK-V1 page replacement algorithm variants for the *TPC-C* benchmark on 100 warehouses

pool sizes ≥500 MB show that this could not improve the performance for the tested workload. The higher overhead of these DGCLOCK-V1 variants results in a lower transaction throughput even for the 100 MB buffer pool. The slightly lower miss rate and slightly higher transaction throughput of the *FixFull* variants indicate that pages selected for eviction by the DGCLOCK-V1 page replacement algorithm, which cannot be evicted temporarily, are hot and should be kept longer in the buffer pool.

The *FixFull* variant is used for the performance evaluation in section 1.3.

1.2.11.2 DGCLOCK-V2

Introduction The only difference between the DGCLOCK-V2 page replacement algorithm and its "static" counterpart GCLOCK-V2 is that instead of one parameter R for all pages, there are many *parameters R_i for different page types i*. A larger value R_i for a specific page type allows prioritizing this page type i in the buffer pool.

Updates of Page Reference Statistics The DGCLOCK-V2 page replacement algorithm *sets* the corresponding usage-count ($UC(p) = R_p$) of a page

p to the value of the parameter R_p, which is specific to the page type of p, on page hit (or alternatively on page unfix). Page types i with greater values of R_i are prioritized and therefore pages of that type are evicted later.

Performance Evaluation Figure 1.22 shows transaction throughput and figure 1.23 shows the miss rate achieved with different variants of the DGCLOCK-V2 page replacement algorithm for different buffer pool sizes. Details of the benchmark setup can be found in section 1.3.

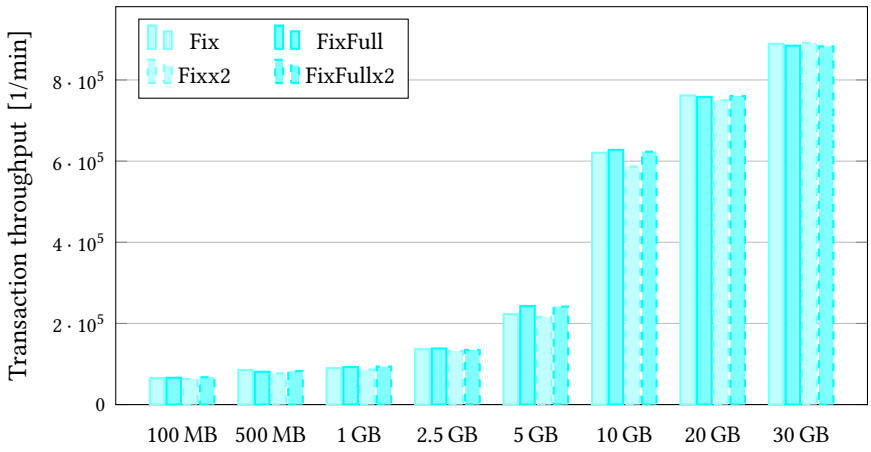


Figure 1.22: Transaction throughput of the DGCLOCK-V2 page replacement algorithm variants for the *TPC-C* benchmark on 100 warehouses

The evaluated variants of DGCLOCK-V2 and the corresponding parameter values are the same as for DGCLOCK-V1.

The miss rates achieved by the different DGCLOCK-V2 variants only vary for very small buffer pools. There, the variants that do not update the usage-count when a page cannot be evicted temporarily achieve lower miss rates. However, the lower miss rates of these variants do not lead to a higher transaction throughput. The differences in transaction throughput are small for larger buffer pools. For larger buffer pools, the variants of the DGCLOCK-V2 page replacement algorithm that end in *x2* show slightly worse transaction throughput than the other variants. The *FixFull* variant performs best for all these buffer pool sizes.

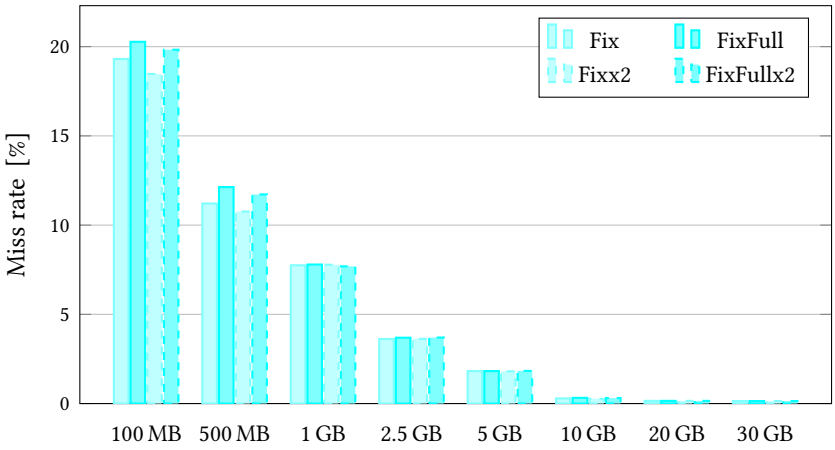


Figure 1.23: Miss rate of the DGCLOCK-V2 page replacement algorithm variants for the *TPC-C* benchmark on 100 warehouses

The higher overhead of the *x2* variants results in lower transaction throughput for any buffer pool size. The slightly higher transaction throughput of the *FixFull* variants indicates that pages selected for eviction by the DGCLOCK-V2 page replacement algorithm that cannot be evicted temporarily are hot, but the higher miss rate of these variants for small buffer pools indicates that most other pages in such a small buffer pool are hotter for the tested workload.

The *FixFull* variant is used for the performance evaluation in section 1.3.

1.2.12 Least Reference Density (LRD)

Introduction The LRD page replacement policy evicts pages based on the so-called *reference density*—i.e. the *number of references to a page per total number of references* to all pages in the buffer pool since that page was fetched. This policy, like GCLOCK and DGCLOCK, has been—proposed by Effelsberg and Härder in [EH84].

Page Reference Statistics The page reference statistics used by the LRD page replacement policy consist of a *global reference counter* and, for

each buffered page, a *reference counter*, the value of the global reference counter at the *time of the first reference*, and—to allow concurrent eviction of pages by multiple working threads—a flag indicating whether another thread has *already selected* the buffer frame for eviction.

Exemplary page reference statistics for a buffer pool with sixteen buffer frames are shown in figure 1.24. It shows for each buffer frame the reference counter (e.g., 120 for buffer frame 0), the value of the global reference counter when the contained page was fetched (e.g., 2450 for buffer frame 5), and the flag indicating whether a working thread has already selected that frame as a candidate for eviction (true only for buffer frame 13). The current reference density of each buffer frame is calculated in the rightmost column. A working thread that is currently evicting pages has obviously already found the page with the *lowest reference density*—it is contained in buffer frame 13.

	Reference count	First reference	Already selected	Global ref. count: 2500
0	120	0	false	$\frac{125}{2500-0} = 0.05$
1	4	1500	false	$\frac{4}{2500-1500} = 0.004$
2	7	600	false	$\frac{7}{2500-600} \approx 0.0037$
3	2	2100	false	$\frac{2}{2500-2100} = 0.005$
4	8	400	false	$\frac{8}{2500-400} \approx 0.0038$
5	1	2450	false	$\frac{1}{2500-2450} = 0.02$
6	3	1800	false	$\frac{3}{2500-1800} \approx 0.0043$
7	5	1150	false	$\frac{5}{2500-1150} \approx 0.0037$
8	1	2425	false	$\frac{1}{2500-2425} \approx 0.0133$
9	1	2400	false	$\frac{1}{2500-2400} = 0.01$
10	2	2200	false	$\frac{2}{2500-2200} \approx 0.0067$
11	5	1100	false	$\frac{5}{2500-1100} \approx 0.0036$
12	6	1000	false	$\frac{6}{2500-1000} = 0.004$
13	4	1200	true	$\frac{4}{2500-1200} \approx 0.0031$
14	8	400	false	$\frac{8}{2500-400} \approx 0.0038$
15	9	200	false	$\frac{9}{2500-200} \approx 0.0039$

Figure 1.24: The page reference statistics of the LRD-V1 and LRD-V2 page replacement algorithms

Page Eviction The algorithm used by the LRD page replacement policies to select an eviction candidate is shown in Algorithm 1.14. The computational complexity of this algorithm *growns linearly with the number of buffer frames*, making it a performance bottleneck in almost every modern OLTP application.

The algorithm *searches linearly* in the buffer pool for the page with the least reference density. minIndex is the index of the buffer frame with the least reference density among those already compared. The reference density of the page in the buffer frame minIndex is stored in minReferenceDensity (must be ∞ at the beginning) and to be able to verify at the end that the page in the buffer frame minIndex was not *referenced again during the search*, the value of the reference counter used for calculating the minReferenceDensity is stored in minReferences.

```
 1: function SELECT
 2:     minIndex ← −1
 3:     minReferences ← 0
 4:     minReferenceDensity ← ∞
 5:     while true do
 6:         for i ← 0 to maxBufferIndex do
 7:             if (referenceCount[i] / (globalReferenceCount − firstReference[i])) < minReferenceDensity then
 8:                 if NOT alreadySelected[i].TEST_AND_SET then
 9:                     alreadySelected[minIndex].CLEAR
10:                     minIndex ← i
11:                     minReferences ← referenceCount[i]
12:                     minReferenceDensity ← referenceCount[i] / (globalReferenceCount − firstReference[i])
13:                 end if
14:             end if
15:         end for
16:         if minReferences ≠ referenceCount[minIndex] then
17:             alreadySelected[minIndex].CLEAR
18:             minReferenceDensity ← ∞
19:             continue
20:         else
21:             return minIndex
22:         end if
23:     end while
24: end function
```

Algorithm 1.14: Selection of eviction candidates by the LRD-V1 and LRD-V2 page replacement strategies

A working thread searching for a page with the least reference density will always *flag* the buffer frame minIndex as already selected to ensure that

multiple working threads concurrently searching for an eviction candidate will find *unique pages*. The flag is also used to exclude *unused buffer frames*—e.g. explicitly evicted ones—from this process. If the found page *temporarily cannot be evicted*, the flag is removed again, and to prevent *an infinite loop* in which this page would be selected over and over again, its reference counter is incremented (but not the global reference counter). The reference counter corresponding to a page that *cannot be evicted permanently* is set to the current value of the global reference counter to delay the next selection for eviction. If the maximum value of the used integer data type was used, the next reference to this page would cause an overflow of the reference counter.

1.2.12.1 LRD-V1

Introduction The LRD-V1 page replacement strategy is more basic than the LRD-V2 page replacement strategy, which works with aging of the reference statistics. In this version of the LRD page replacement strategy, each reference to a page since the last time the page was fetched into the buffer pool is valued equally.

Updates of Page Reference Statistics When a page is *fetched* into the buffer pool, its corresponding *first reference* value is set to the current value of the *global reference count*, the value of the *global reference count* is incremented, and the *reference count* of the page is set to 1. The corresponding buffer frame is marked as not already selected.

In the event of a *page hit*, both the corresponding *reference count* and the *global reference count* will be incremented. The page is marked as not already selected for eviction.

If a page selected for eviction is *temporarily impossible to evict*, its *reference count* is incremented so that it is not selected over and over again. It is marked as not already selected for eviction.

If a page selected for eviction *cannot be evicted permanently*, its *reference count* is set to the current value of the *global reference count* to prevent further selection for eviction.

1.2.12.2 LRD-V2

Introduction The LRD-V2 page replacement strategy adds *aging of the reference statistics* to the LRD-V1 page replacement strategy. In this way, pages that are referenced very frequently for a limited period of time are evicted more quickly once they are not referenced again. In this way, the *reference frequency* becomes less important than the *reference recency*.

Updates of Page Reference Statistics LRD-V2 applies the *same page reference statistics* updates as LRD-V1, but after every f^{th} global reference an *aging function* is applied to the *reference count*s of all buffer frames. Simple aging function are e.g. subtraction of a fixed value x (referenceCount = $\max(0, \text{referenceCount} - x)$) or multiplication by a fixed factor $x < 1$ (referenceCount = referenceCount $\cdot\, x$).

Performance Evaluation Figure 1.25 shows transaction throughput and figure 1.26 shows the miss rate achieved with different variants of the LRD-V2 page replacement algorithm for different buffer pool sizes. Details of the benchmark setup can be found in section 1.3.

The variants of the LRD-V2 page replacement policy are named according to the scheme $f \circ x$ with the *aging period* f, the *aging function* $\circ$ and the *aging operand* x. For a buffer pool with b buffer frames, a aging period of f means that the specified aging function with the given aging operand is applied to each *reference count* after $b \cdot f$ global page references. For example, the $2 \cdot 0.75$ variant multiplies after every $2 \cdot b$ page references each *reference count* by 0.75. The variant $2 - 10$ subtracts 10 from each *reference count* (minimum value 0) with the same period.

The *most striking result* of the performance evaluation of LRD-V2 is that as long as around <10 % of the initial DB fits in the buffer pool, the transaction throughput for all variants of the replacement policy shrinks as the buffer pool size increases. Despite the significant growth in page hit rate, some variants such as $2 \cdot 0.5$ perform better on a small 100 MB buffer pool than on a large 5 GB one, which is very surprising. In contrast, transaction throughput increases again with buffer pool size for larger buffer pools, which is in accordance with intuition.

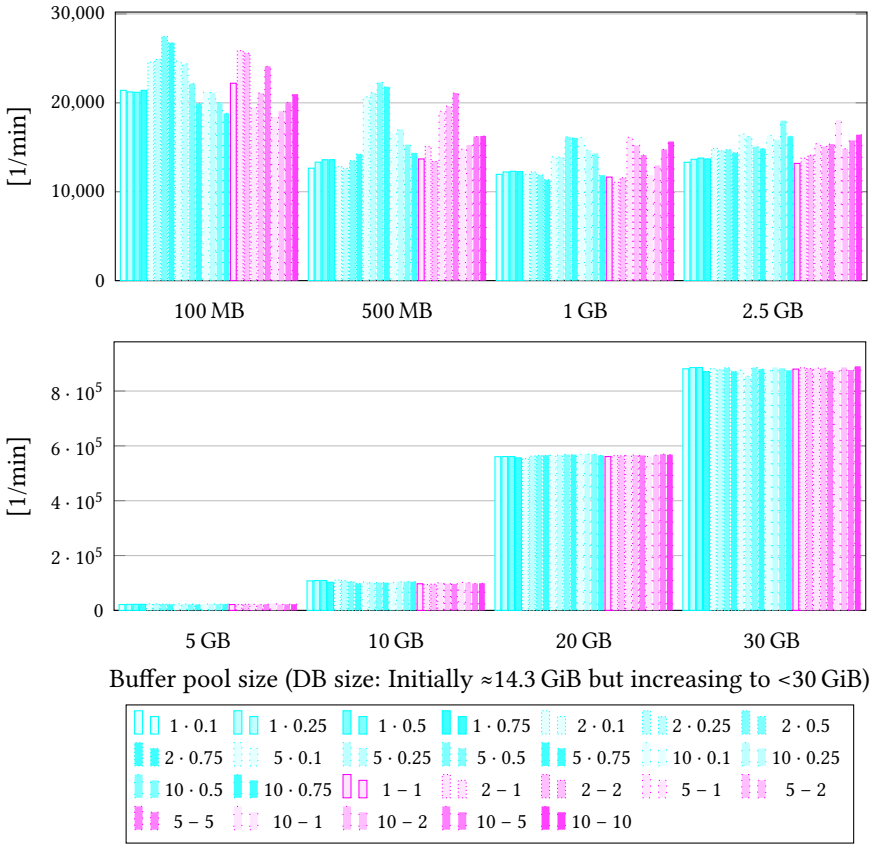


Figure 1.25: Transaction throughput of the LRD-V2 page replacement algorithm variants for the *TPC-C* benchmark on 100 warehouses

The *optimal aging frequency* with respect to transaction throughput *decreases with an increasing size of the buffer pool.* For a buffer pool size of 100 MB, the variants with an aging period of $f = 2$ perform best, but for one with 2.5 GB, the variants with an aging period of $f = 10$ are better. This behavior is not only the result of greater overhead from the aging process with larger buffer pools, but is also supported by the hit rates. For each buffer size, the faster LRD-V2 variants are also those with a lower page miss rate, while those with a high miss rate always perform poorer.

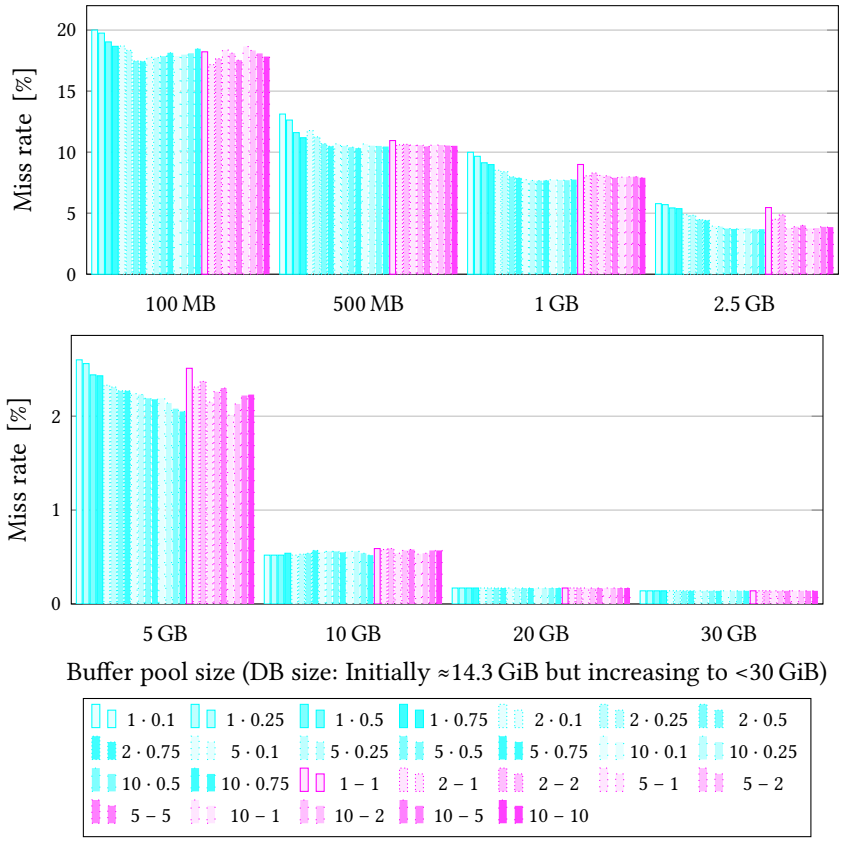


Figure 1.26: Miss rate of the LRD-V2 page replacement algorithm variants for the *TPC-C* benchmark on 100 warehouses

For *growing aging periods*, decreasing factors for multiplicative aging and increasing subtrahents for subtractive aging are required to achieve the best transaction throughput. It keeps the influence of aging stable, the *reference count* values need to *age more strongly* with *less frequent aging*.

For *larger buffer pools* (≥5 GB), the *miss rates* and *transaction throughput* of the different variants of the LRD-V2 page replacement policy are almost identical. The differences are statistically insignificant.

Due to its decent performance for all buffer pool sizes the 10 – 10 variant is used for the performance evaluation in section 1.3.

1.2.13 Least Frequently Used (LFU)

Introduction The LFU page replacement policy is one of the *oldest* approaches to page replacement. It *considers only the reference frequency* when deciding on an eviction candidate. The idea behind this criterion is that hot pages are referenced very frequently.

However, one problem with this page replacement policy is that pages that were referenced very frequently remain in the buffer pool forever, even if they left the working set a long time ago (violation of the *five-minute rule*). And because the least frequently referenced page is very often also the *least recently fetched page*, the LFU page replacement policy has the same problem as the FILO page replacement policy—the page selected for eviction is often still fixed and therefore temporarily impossible to evict.

Page Reference Statistics The page reference statistics maintained by the LFU page replacement policy consists of one *reference count* per buffer frame. And to allow concurrent eviction of pages by multiple working threads, there is also flag for each buffer frame indicating whether another thread has *already selected* the contained page for eviction.

Updates of Page Reference Statistics When a page is *fetched* into the buffer pool, the corresponding reference count is set to 1 and the corresponding buffer frame is marked as not already selected. Whenever this page is *referenced again*, the reference count is incremented by 1.

If a page selected for eviction is *temporarily impossible to evict*, its *reference count* is incremented so that it is not selected over and over again. It is marked as not already selected for eviction.

If a page selected for eviction *cannot be evicted permanently*, its *reference count* is set to half the maximum value of the data type used, both to prevent further selection for eviction and to ensure that there is no overflow to 0 the next time the page is referenced.

Page Eviction The algorithm used by the LFU page replacement policy to select an eviction candidate is shown in Algorithm 1.15. It is similar to the one used by the LRD page replacement policies, and therefore its computational complexity *grows linearly with the number of buffer frames*.

The algorithm *searches linearly* in the buffer pool for the page with the least reference frequency. minIndex is the index of the buffer frame with the least reference frequency among those already examined. The reference frequency of the page in the buffer frame minIndex is stored in minReferences (must be ∞ at the beginning). By comparing minReferences with the current reference count of the buffer frame minIndex, it is verified at the end that the page in the buffer frame minIndex was not *referenced again during the search*.

```
 1: function SELECT
 2:     minIndex ← −1
 3:     minReferences ← ∞
 4:     while true do
 5:         for i ← 0 to maxBufferIndex do
 6:             if referenceCount [i] < minReferences then
 7:                 if NOT alreadySelected [i] .TEST_AND_SET then
 8:                     alreadySelected [minIndex] .CLEAR
 9:                     minIndex ← i
10:                     minReferences ← referenceCount [i]
11:                 end if
12:             end if
13:         end for
14:         if minReferences ≠ referenceCount [minIndex] then
15:             alreadySelected [minIndex] .CLEAR
16:             minReferences ← ∞
17:             continue
18:         else
19:             return minIndex
20:         end if
21:     end while
22: end function
```

Algorithm 1.15: Selection of eviction candidates by the LFU page replacement strategy

A working thread searching for a page with the least reference frequency will always *flag* the buffer frame minIndex as already selected to ensure that multiple working threads concurrently searching for an eviction candidate will find *unique pages*. The flag is also used to exclude *unused buffer frames*—e.g. explicitly evicted ones—from this process. If the found page *temporarily cannot be evicted*, the flag is removed again, and to prevent *an infinite loop* in which this page would be selected over and over again, its reference counter is incremented. The reference counter corresponding to a page that *cannot be evicted permanently* is set to half the maximum value of the data type used. If the maximum value of the used integer data type was

used, the next reference to this page would cause an overflow (to 0) of the reference counter.

1.2.14 LFU With Dynamic Aging (LFUDA)

Introduction The LFUDA page replacement policy is an extension of the LFU page replacement policy that was proposed by Arlitt et al. in [Arl+00]. It tries to *solve the main problem of the LFU* page replacement policy—the pollution of the buffer pool with pages that were referenced very frequently, possibly a long time ago. The LFUDA page replacement policy does this by setting the initial reference count of a page just fetched to a *dynamically growing inflation factor* instead of 1—thus the reference count of pages just fetched can exceed the value of older pages that were very frequently fetched.

Page Reference Statistics In addition to the page reference statistics maintained by the LFU page replacement algorithm (reference counts and "already selected" flags per buffer frame), the LFUDA page replacement algorithm also manages the *inflation factor*, which is updated each time a page is selected for eviction.

Updates of Page Reference Statistics When a page is *fetched* into the buffer pool, the corresponding reference count is set to the *current value of the inflation factor* and the corresponding buffer frame is marked as not already selected. In all other situations, the LFUDA page replacement algorithm updates its page reference statistics in the same way as the LFU page replacement algorithm.

Page Eviction The LFUDA page replacement algorithm selects eviction candidates in the same manner as the LFU page replacement algorithm (Algorithm 1.15), but before returning the buffer index of the eviction candidate, it *sets the inflation factor to the reference count of the eviction candidate.*

1.2.15 LeanStore Replacement

Introduction The LeanStore page replacement algorithm was proposed by Leis et al. in [Lei+18]. It is based on pointer swizzling in the buffer pool proposed by Graefe et al. in [Gra+14] and selects eviction candidates based on randomly unswizzled pages that have not been recently used. With this technique it approximates the behavior of the ZCLOCK page replacement algorithm. In most cases, there is no need to update the page reference statistics on page hits.

Page Reference Statistics The LeanStore page replacement algorithm *does not keep page reference statistics in the usual sense for each page.* Instead, it uses a FIFO queue—the *cooling stage*—which contains a fraction of the buffer indexes that refer to pages that have not been recently referenced. To allow multiple working threads to enqueue and dequeue buffer frame indexes concurrently, a latch (mutex) must be acquired before each operation on the cooling stage. It also uses the information—stored in the pointer to a page contained in the parent page (within the B tree) of that page—*whether that page is swizzled* (for a detailed description of the pointer swizzling technique, see subsection 2.4.1). The buffer frame containing a page that is not swizzled is included in the cooling stage.

As an additional optimization, it flags pages that cannot be evicted permanently as "not evictable".

Updates of Page Reference Statistics When a page is *fetched* into the buffer pool, the corresponding pointer within the parent page will usually be swizzled. Accordingly, the corresponding buffer frame index is not added to the cooling stage.

But *at a random point in time*, this pointer (if the page is not marked as "not evictable") gets unswizzled during the page eviction of other pages, and therefore enters the cooling stage. If that page is then *re-referenced* (*page hit*) after the unswizzling, it is removed from the cooling stage and its pointer within the parent page is again swizzled. In other cases, no update of the page reference statistics is required when a *page hit* occurs.

If a page selected for eviction *cannot be evicted permanently*, it is flagged "not evictable".

Page Eviction The algorithm used by the LeanStore page replacement algorithm to fill the cooling stage and select an eviction candidate is provided in Algorithm 1.16.

As soon as the *cooling stage is half empty*, it is refilled until it contains maxCoolingStageSize buffer frame indexes. The working threads do this before they start to reclaim buffer frames. To *add a page to the cooling stage*, a random buffer frame index is picked—the cooling candidate—, an attempt is made to unswizzle the pointer to the contained page, and on success, the cooling candidate is enqueued to the cooling stage.

More specifically, this means that once a working thread has *randomly selected* a buffer frame using the SplitMix32 PRNG, it checks if this cooling candidate is already in the cooling stage or if the contained page cannot be evicted. If the buffer frame *could be reclaimed* and is *not already in the cooling* stage, the working thread tries to *acquire the latch* of the cooling candidate in *exclusive mode*. If the latch is held by another working thread, the contained page is probably not cold and should therefore not be added to the cooling stage—another cooling candidate must be picked at random. Now, in order to unswizzle the pointer in the *parent page*, the corresponding buffer frame must be *latched in shared mode* to ensure that the page is not changed until the pointer is unswizzled. If the latch cannot be acquired, the unswizzling cannot be performed, and therefore another cooling candidate must be picked randomly. However, if both latches—the one for the cooling candidate and the one for the parent page—have been acquired, the *pointer will be unswizzled*. If successful, the buffer index of the cooling candidate is enqueued in the cooling stage queue and the latches are released. Otherwise, the latches must be released and another cooling candidate must be randomly picked.

If the cooling stage contains enough pages, the page that has been *in the cooling stage for the longest time* (FIFO) is taken out of the queue and returned as an *eviction candidate*. If it *cannot be evicted temporarily*, it is again enqueued to the cooling stage queue for a later retry.

Performance Evaluation A large cooling stage—containing a larger fraction of the buffer frame indexes when full—results in a lot of unnecessary unswizzling and swizzling operations. But if the cooling stage is too small,

```
1: function SELECT
2:     while true do
3:         if |coolingStage| < maxCoolingStageSize/2 then
4:             while |coolingStage| < maxCoolingStageSize do
5:                 coolingCandidate ⟵ r randomly ∈ {x ∈ ℕ‖0 ≤ x ≤ maxBufferIndex}
6:                 coolingStageLatch.ACQUIRE
7:                 if |coolingStage| ≥ maxCoolingStageSize then
8:                     coolingStageLatch.RELEASE
9:                     continue
10:                end if
11:                if coolingCandidate ∈ coolingStage ∨ notEvictable [coolingCandidate] then
12:                    coolingStageLatch.RELEASE
13:                    continue
14:                else
15:                    coolingStageLatch.RELEASE
16:                    if ¬ISSWIZZLED(coolingCandidate) ∨ ISLATCHED(coolingCandidate) then
17:                        continue
18:                    end if
19:                    if ¬ACQUIRELATCH(coolingCandidate, X) then
20:                        continue
21:                    end if
22:                    if ¬ISEVICTABLE(coolingCandidate) then
23:                        RELEASELATCH(coolingCandidate)
24:                        continue
25:                    end if
26:                    coolingCandidateParent ⟵ GETPARENT(coolingCandidate)
27:                    if ¬ACQUIRELATCH(coolingCandidateParent, S) then
28:                        RELEASELATCH(coolingCandidate)
29:                        continue
30:                    end if
31:                    if UNSWIZZLE(coolingCandidateParent) then
32:                        coolingStageLatch.ACQUIRE
33:                        coolingStage.ENQUEUE(coolingCandidate)
34:                        coolingStageLatch.RELEASE
35:                        RELEASELATCH(coolingCandidateParent)
36:                        RELEASELATCH(coolingCandidate)
37:                    else
38:                        RELEASELATCH(coolingCandidateParent)
39:                        RELEASELATCH(coolingCandidate)
40:                        continue
41:                    end if
42:                end if
43:            end while
44:        end if
45:        coolingStageLatch.ACQUIRE
46:        evictionCandidate ⟵ coolingStage.DEQUEUE
47:        coolingStageLatch.RELEASE
48:        if evictionCandidate ≠ NULL then
49:            return evictionCandidate
50:        else
51:            continue
52:        end if
53:    end while
54: end function
```

Algorithm 1.16: Selection of eviction candidates by the LeanStore page replacement algorithm

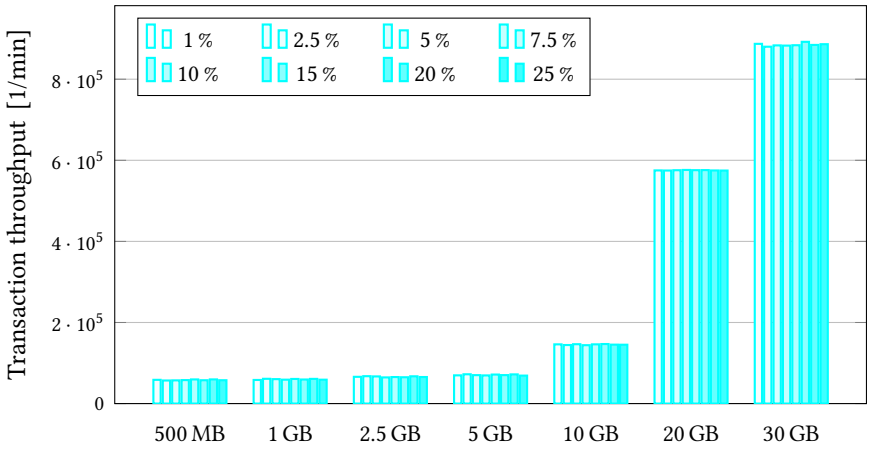


Figure 1.27: Transaction throughput of the LeanStore page replacement algorithm variants for the *TPC-C* benchmark on 100 warehouses

pages are evicted very quickly after being added to the cooling stage. That way, the cooling stage cannot be used to verify that a page is not used. For this reason, the performance evaluation of the LeanStore page replacement algorithm covers cooling stage sizes from 1 % of the buffer pool up to 25 % thereof.

Figure 1.27 shows transaction throughput and figure 1.28 shows the miss rate achieved with different variants of the LeanStore page replacement algorithm for different buffer pool sizes. Details of the benchmark setup can be found in section 1.3.

Neither the *miss rate* nor the *transaction throughput* differ significantly for different cooling stage sizes. Neither the pointer (un)swizzling overhead for larger cooling stage sizes nor the less detailed page reference statistics for smaller ones have a significant impact.

While any variant could have been used for the performance evaluation in section 1.3, the one with a cooling stage size of 2.5 % of the buffer pool was chosen.

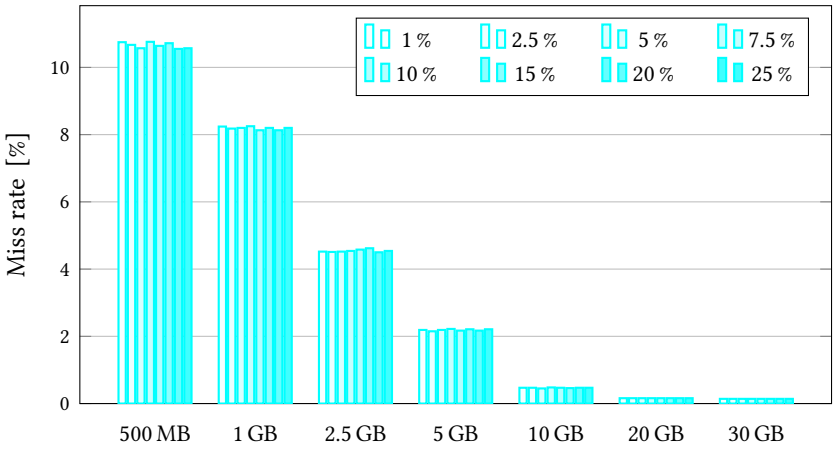


Figure 1.28: Miss rate of the LeanStore page replacement algorithm variants for the *TPC-C* benchmark on 100 warehouses

1.3 Performance Evaluation

As described in the introductory section (section 1.1) of this chapter, the page replacement algorithm used in a DBMS has a *significant impact on system performance*. There are three key *performance factors* of a page replacement algorithm:

1. achieved hit rate
2. overhead on page hit (e.g. waiting on a latch)
3. overhead on page miss (selecting an eviction candidate)

When page replacement algorithms are compared in *scientific papers*, the focus is usually on the 1st of the performance factors—the hit rate. [Bél66], [Smi78], [EH84], [OOW93], [KLW94], [JS94], [AFJ00], [Arl+00], [JZ02], [MM03], [BM04], [Li18] and [Li19] compare only the hit rates achieved with different page replacement algorithms in different situations—for example for different applications or buffer pool sizes. [JCZ05], [ZPL01], and [Li18] give some limited data on overall system performance for different page replacement algorithms and [Cor69] compares the hit rate and CPU overhead for different parameterizations of one page replacement algorithm.

There are *many reasons* why the overhead of page replacement algo-

rithms is so rarely compared. Hit rates are usually measured in a *simulated buffer pool* with given page reference strings—which are either synthetically generated or recorded from real software runs. In this way, the hit rates achieved with the reference behavior of *many different applications* (for which a newly proposed page replacement algorithm is usually to be evaluated) can be compared without having to implement the page replacement algorithms in different applications. Furthermore, it is usually *easier to implement* a page replacement algorithm for a simulated buffer pool, since fewer exceptional cases have to be considered. The results of these simulations are also easier to compare because they *abstract from real-world influences* such as the system configuration and the elapsed real time between successive page references. But ignoring factors such as the overhead of page hit synchronization, which occurs, for example, in the Hash-Map-Doubly-Linked-List implementation of the LRU page replacement algorithm, results in a biased—wrong—*ranking of page replacement algorithms.*

To allow a more realistic estimation of the achievable performance of the page replacement algorithms described in section 1.2, this section provides a *combined comparison of the hit rate and transaction throughput* achieved with the different algorithms.

1.3.1 System Configuration

C++ Compiler GCC (GNU Compiler Collection) 7.5.0 [6][7]
OS GNU/Linux 4.15.0 [8]
Kernel Ubuntu 18.04 LTS (Bionic Beaver) [9]
CPU 2× Intel® Xeon® X5670 @12 × 2.93GHz released early 2010
RAM 12 × 8GiB (96 GiB) Samsung DDR3-SDRAM @1333 MHz[10]
Storage 2 × 256 GiB Samsung SSD 840 PRO[11][12]

[6] https://gcc.gnu.org/gcc-7/
[7] Used flags: -g -O3 -fexpensive-optimizations -finline-functions -flto -fno-fat-lto-objects -fno-strict-aliasing -march=native
[8] https://mirrors.edge.kernel.org/pub/linux/kernel/v4.x/
[9] https://releases.ubuntu.com/18.04/
[10] https://www.samsung.com/semiconductor/dram/module/M393B1K70CH0-CH9/
[11] https://www.samsung.com/at/support/model/MZ-7PD256BW/
[12] One SSD is only used for the DB file, the other is only used for the transaction log.

1.3.2 Benchmark

The benchmark used for this performance evaluation is the well recognized OLTP benchmark *TPC-C*[13]. *TPC-C* simulates a classical business application for an OLTP system—a *wholesale supplier*. It is the successor of the much simpler and deprecated *TPC-A*[14] and the predecessor of the much more sophisticated *TPC-E*[15].

Database The DB consists of nine tables. The size of the database can be scaled using the *scaling factor* W, which allows the benchmark to simulate many different *business sizes*. With a scaling factor of $W = 1$ the initial database size is ≈ 143 MiB.

W stands for "warehouse". The simulated wholesale supplier operates exactly one warehouse if $W = 1$ is used. This warehouse has 100 000 items in stock and covers 10 districts with 3000 customers each. Per customer an order history of ≥1 orders is managed. Open orders consist of 5–15 line items from the warehouse stock and as soon as an open order is processed, it is added to the order history without details such as line items.

There are text fields with variable (SQL VARCHAR) and fixed length (SQL CHAR), fields with signed numeric (SQL DECIMAL), date and time (SQL TIMESTAMP) and ID data types (can be implemented with SQL INT). The NEW ORDER table consists of 3 ID fields and each tuple has a size of 6 B. The CUSTOMER table consists of 22 fields of different types and each tuple is ≤679 B. The tuple sizes of the seven other tables are between these two tuple sizes. There are *primary indexes* based on the (composite) primary keys, and two tables also have *secondary indexes*.

Transactions A number of simulated *online terminal sessions* are used to concurrently submit transactions to the database. *12 terminal sessions* (the number of physical cores available) were used for this performance evaluation. The benchmark defines five types of transactions that simulate different activities of a wholesale supplier. The transactions must meet the

[13] http://www.tpc.org/tpcc/
[14] http://www.tpc.org/tpca/
[15] http://www.tpc.org/tpce/

ACID properties. The data accesses by these transactions follow certain non-uniform distributions that lead to *contention on the data*.

45 % of the transactions submitted to the system are NEW ORDER transactions—characterized as mid-weight read/write transactions. The light-weight read/write transaction PAYMENT is used for 43 % (+4 %, because the implementation included with the used prototype DBMS *Zero*[16] also runs PAYMENT instead of DELIVERY) of the transactions submitted to the system. The only two read-only transactions defined by the *TPC-C* specification—the mid-weight ORDER STATUS transaction and the heavy STOCK LEVEL transaction—each account for only 4 % of the transactions making *TPC-C update-focused*.

1.3.3 Limitations of this Performance Evaluation

The configuration of the *server system* used for the performance evaluation, the properties of the *database prototype* used and the *benchmark* limit the generalizability of the results.

A 10 year old server with two SSDs connected via SATA and DDR3 SDRAM is no longer a typical database server today. Even cheap servers use CPUs with much more cores and CPU cache. Midrange servers often have multiple TiB DDR4 SDRAM and use much faster NVMe SSDs. Even faster NVRAM solutions for DIMM slots are also available[17] at a much higher price.

The DBMS prototype *Zero*[18][19] supports storing data only in *Foster B trees* with one B-tree per primary index. Other data structures for indexes such as hash index structures, multidimensional index structures or trie structures are not supported. *Zero*'s buffer pool uses a *global buffer allocation* and only *demand paging*. Hints about pages that may be used in the future by a running transaction, which could be used for prefetching, or hints about the page reference pattern (e.g. loop) of a particular transaction, which could be used for page eviction decisions, are not supported.

[16] https://github.com/iMax3060/zero

[17] https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf

[18] https://github.com/iMax3060/zero

[19] See my Bachelor's thesis [Gil17] for more details about the history of Zero.

Zero is an embedded DBMS, not a relational DBMS. The *logical data structures* layer, which provides a set-oriented interface, e.g. using a query language like SQL, is missing in the system. The *logical access paths* layer is also not part of *Zero* itself, but implemented as part of its benchmark framework. Therefore, *Zero* is not representative for most DBMSs used for DBs larger-than-memory.

The benchmark used—*TPC-C*—was specified in 1992, and *applications for OLTP have diversified dramatically* over the past nearly 30 years. Especially the web with the advent of e-commerce and a hardly assessable number of other online services that are indispensable today leads to many new applications for OLTP and OLAP database systems. While the traditional business applications for OLTP applications still exist almost unchanged but the new applications are very often *more read-heavy* than *TPC-C*. And it is almost impossible to specify a benchmark that replicates the diversity of *Big Data* applications.

1.3.4 Results

Figure 1.29 shows the transaction throughput and figure 1.30 shows the miss rate achieved with all the evaluated page replacement algorithms for different buffer pool sizes.

FILO, MRU and the Hash-Map-Doubly-Linked-List implementation of LRU-K do not work for small buffer pools. The latter two even stalled the DBS if the buffer pool is not large enough for the entire DB.

The Hash-Map-Doubly-Linked-List implementation of LRU—called List-LRU here—and SLRU do not scale with respect to the *transaction throughput*. For *larger buffer pool sizes*, they perform ≈94 % worse than the fastest competing page replacement algorithms. Their performance does not increase at all with buffer pool size for buffer pools >10 GB. They are also slower than most of the competition for *smaller buffer pools*, but the gap is smaller—in both absolute and relative terms.

For the buffer pool sizes where the Hash-Map-Doubly-Linked-List implementation of LRU-K works, List-LRU-2, List-LRU-3, and List-LRU-4 achieve almost twice the *transaction throughput* as the Hash-Map-Doubly-Linked-List implementation of LRU. But even the fastest of the three variants—List-LRU-2—is almost 90 % slower than most of the competition.

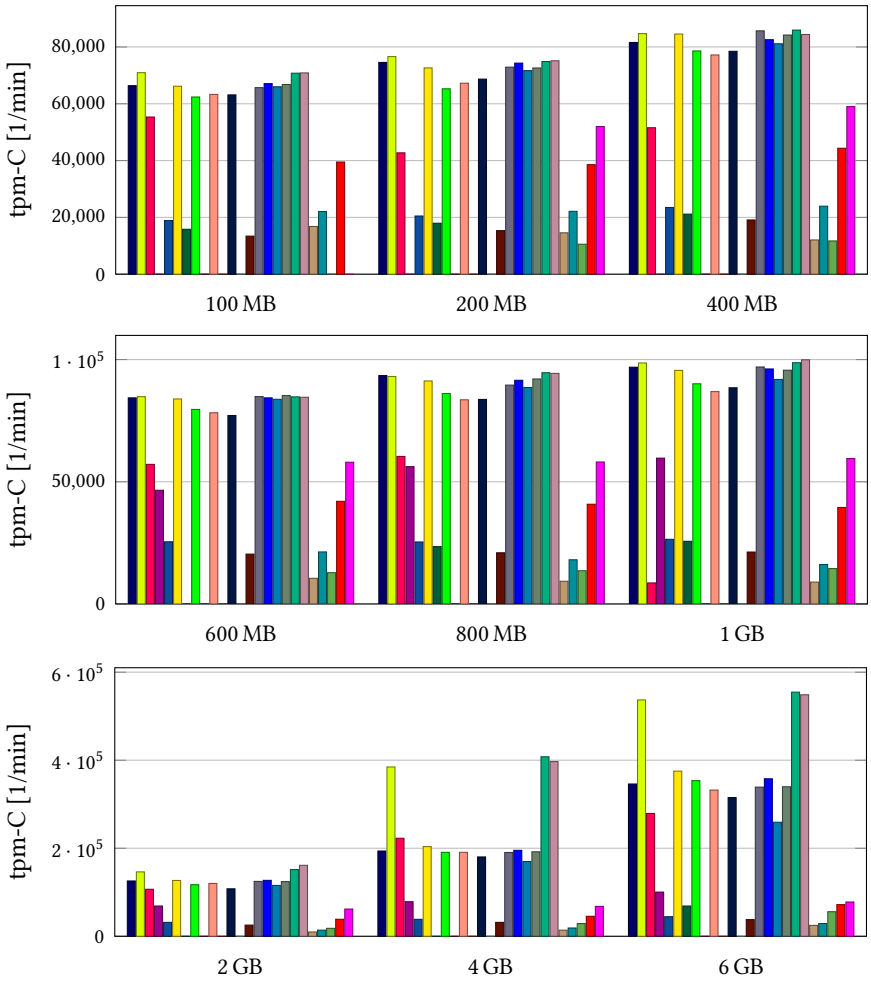


Figure 1.29: Transaction throughput of the page replacement algorithms from section 1.2 for the *TPC-C* benchmark on 100 warehouses

Quasi-FIFO and FILO are in the midrange for *buffer pools smaller than the initial DB* in terms of *transaction throughput*, but they are among the fastest page replacement algorithms when (almost) no pages are evicted. But both algorithms show *random drops* in their *transaction throughput*—Quasi-FIFO

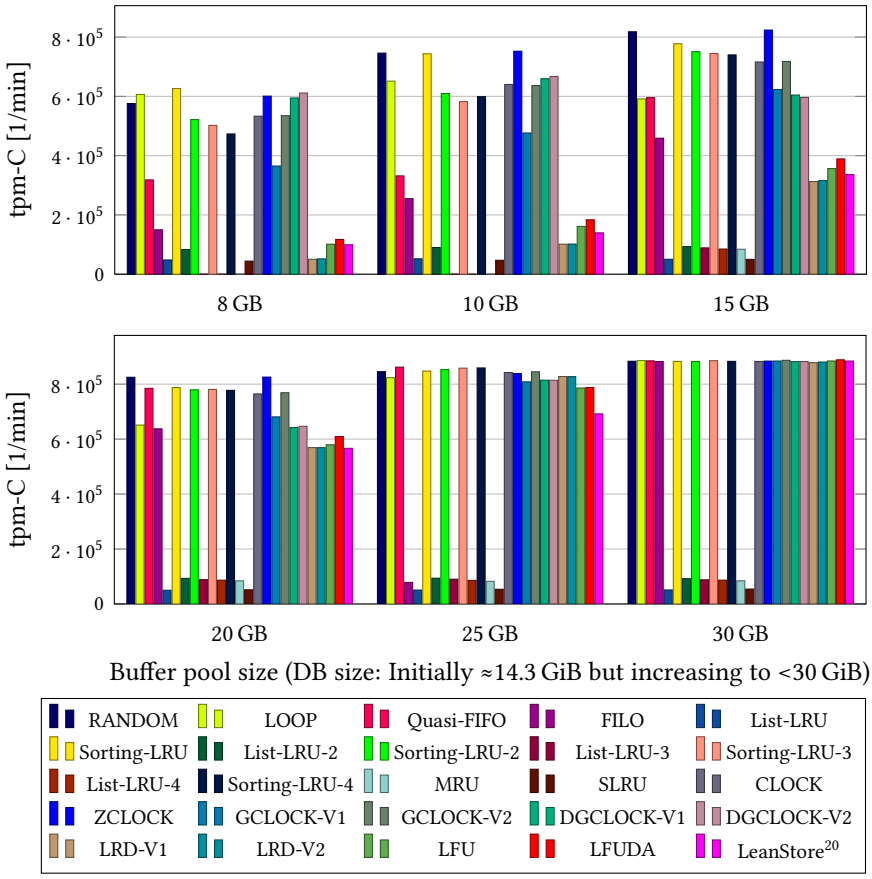


Figure 1.29: Transaction throughput of the page replacement algorithms from section 1.2 for the *TPC-C* benchmark on 100 warehouses (continued)

for a buffer pool with 1 GB size and FILO for one with 25 GB size.

For *very small buffer pools* (≤1 GB), LFU is one of the slowest page replacement policies in the competition—LFUDA is in the mid-range for these buffer pool sizes. Both policies are amongst the slowest for *medium-sized buffer pools* (2 GB–20 GB) and amongst the fastest when the buffer pool has the size of the DB. LFUDA is *constantly faster* than LFU.

[20]LeanStore uses the pointer swizzling technique described in subsection 2.4.1.

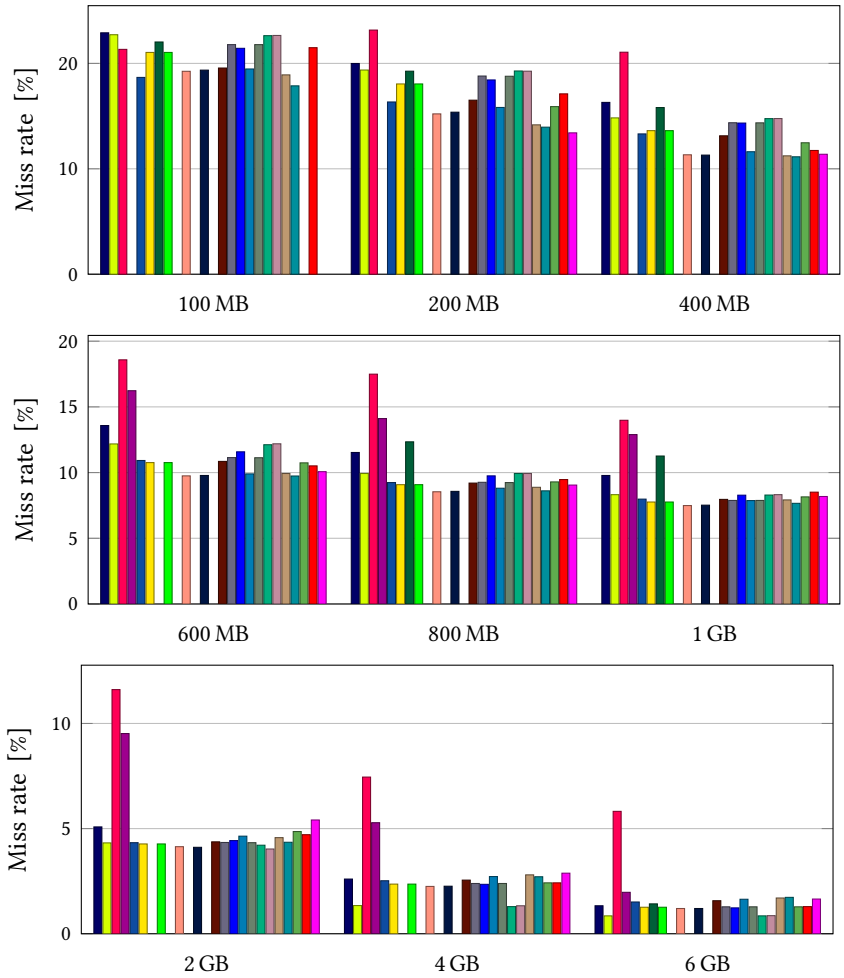


Figure 1.30: Miss rates of the page replacement algorithms from section 1.2 for the *TPC-C* benchmark on 100 warehouses

LRD-V1 and LRD-V2 perform very similar to LFU and LFUDA, but they are consistently slower than at least the latter one.

The Timestamp-Sorting implementation of LRU-K—here referred to as Sorting-LRU-2, Sorting-LRU-3, and Sorting-LRU-4—is almost one of the

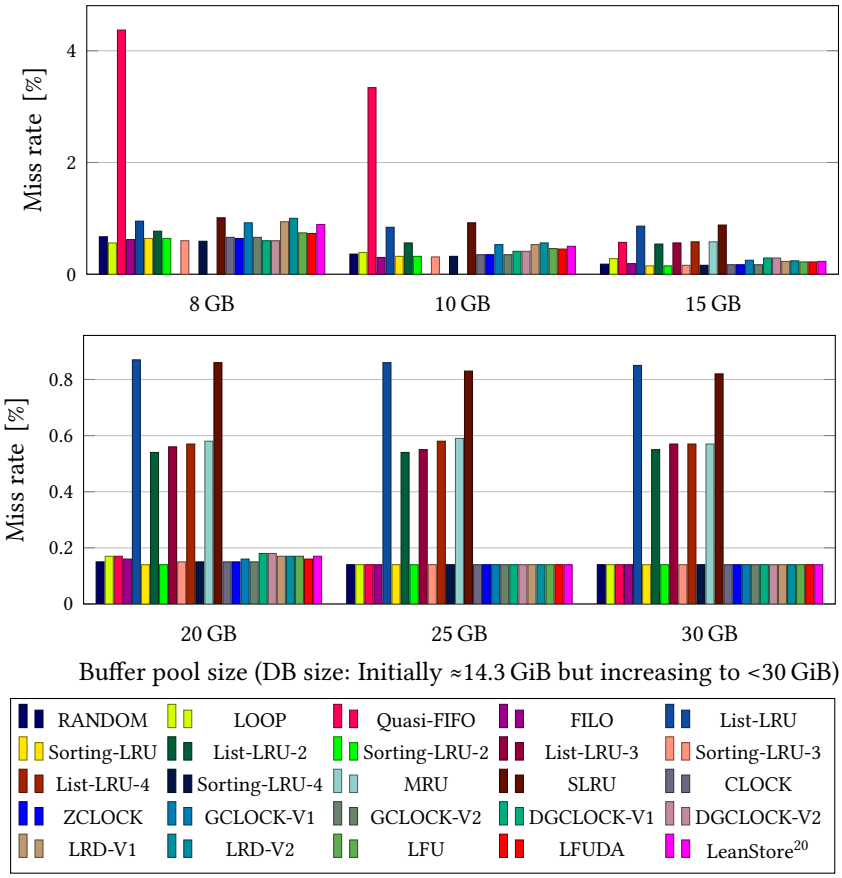


Figure 1.30: Miss rates of the page replacement algorithms from section 1.2 for the *TPC-C* benchmark on 100 warehouses (continued)

fastest page replacement policies in the competition. Except for the smallest buffer pool sizes, Sorting-LRU-2 is the fastest of the three.

RANDOM, the Timestamp-Sorting implementation of LRU, CLOCK, ZCLOCK, GCLOCK-V1 and GCLOCK-V2 are all among the fastest page replacement algorithms for *most of the buffer pool sizes*. But the FIFO implementation LOOP and DGCLOCK-V1 and DGCLOCK-V2 are almost twice as fast for 4 GB and 6 GB buffer pools.

The *transaction throughput* achieved with LeanStore is for *small buffer*

pools of ≤2 GB as decent as that of Quasi-FIFO and for *larger buffer pools* as bad as that of LFU.

Especially for *smaller buffer pool sizes*, the Hash-Map-Doubly-Linked-List implementation of LRU, the Timestamp-Sorting implementation of LRU-K, GCLOCK-V1, LRD-V1 and LRD-V2 achieve a lower *miss rate* than the competition.

For buffer pools smaller than 4 GB, the *miss rate* of FILO is ≈4 % higher than that of most competitors, and Quasi-FIFO maintains an even higher *miss rate* for buffer pools of ≤10 GB size. But once the *initial DB fits in the buffer pool*, both algorithms are among the best in terms of *miss rate*.

The *miss rates* achieved with most page replacement algorithms is below 0.2 % once the *DB fits in the buffer pool*. The responsible page misses are the result of cold starting the DBS—with an empty buffer pool—which is part of each 10 min benchmark run used to gather this data. The page miss rates would converge for a continuously running system to 0 %. The higher page miss rates of approximately 0.9 % of the Hash-Map-Doubly-Linked-List implementation of LRU, the Hash-Map-Doubly-Linked-List implementation of LRU-K, MRU and SLRU are the result of a lower total number of page fixes due to the lower transaction throughput achieved with these page replacement algorithms.

1.3.5 Analysis

The Hash-Map-Doubly-Linked-List implementation of LRU, the Hash-Map-Doubly-Linked-List implementation of LRU-K, MRU, and SLRU scale very poorly regarding *transaction throughput* due to the synchronization required for multithreaded execution. A global latch must be acquired on each page hit—regardless of whether pages are evicted or not. However, this cannot explain the exceptionally high *miss rates* for *buffer pools ≥ 15 GB*.

It is no surprise that the sloppy selection of eviction candidates by LeanStore does not result in high *hit rates*, but although it does not require an update of the page reference statistics on most page hits, the overhead for pointer (un)swizzling is high, keeping the *transaction throughput* relatively low *even with large buffer pools*. The overhead for adding pages to and removing pages from the cooling stage is particularly high when pages are infrequently evicted.

The very low *hit rates* achieved by Quasi-FIFO and especially by FILO are to be expected in consideration of their suboptimal eviction decisions when used with B-trees. But because of the very low overhead (no overhead on page hits), the *transaction throughput* they achieve is very decent.

Considering the expected pollution of the buffer pool with pages referenced very frequently for a short period of time when using the LFU page replacement policy, it reached surprisingly low *miss rates*. However, this may be a result of short (10 min) benchmark runs, where pages that are referenced very frequently remain hot until the end of the benchmark. The simple reference counter results in a competitive *transaction throughput* when the *DB fits into the buffer pool*, but the slow selection of eviction candidates slows down LFU when page evictions actually take place. The lower *miss rate* of LFUDA for *small buffer pools* suggests that some pollution of the buffer pool can be prevented by ageing, but for *bigger buffer pools* the way the dynamic aging of LFUDA works leads to higher *miss rates*. Because of the identical overhead imposed by both algorithms, the *transaction throughput* roughly reflects the differences in *miss rates* between the two algorithms.

The good idea behind the LRD page replacement strategies results in the lowest *miss rates* of all page replacement algorithms for *smaller buffer pools*. However, due to the slow selection of eviction candidates, neither LRD-V1 nor LRD-V2 will achieve high *transaction throughput* when the *DB fits into the buffer pool*. However, the atomic counters that are incremented on a page hit do not cause significant overhead, resulting in a competitive *transaction throughput* for the *biggest buffer pool sizes*.

For the *most buffer pool sizes* (small and large buffer pools), the Timestamp-Sorting implementation of the LRU-K page replacement strategy achieves the lowest *miss rates*. In particular, Sorting-LRU-3 and Sorting-LRU-4 achieve very high *hit rates* for *smaller buffer pools*. For *larger ones*, the Timestamp-Sorting implementation of LRU achieves the same *hit rates*. These high hit rates also lead to a high *transaction throughput*—especially with Sorting-LRU, which has a particularly low overhead. Only few competitors can beat these algorithms in terms of *hit rates* or *transaction throughput*.

The higher *miss rates* of RANDOM and LOOP compared to CLOCK, ZCLOCK, GCLOCK and DGCLOCK are no surprise. But due to the complete lack of overhead and the quick eviction candidate selection, they perform

just as well as the others—for some buffer pool sizes even better.

When comparing CLOCK, ZCLOCK, GCLOCK, and DGCLOCK, they all perform very similarly well on *very small buffer pools*. On *medium size buffer pools*, the ones with more detailed page reference statistics—DGCLOCK-V1 and DGCLOCK-V2—perform better than the others. But low overhead and simplicity are king when the *hit rates are high anyways*, resulting in CLOCK and ZCLOCK outperforming the others on *big buffer pools*. A reason why GCLOCK-V1 is usually slower than GCLOCK-V2, while DGCLOCK-V1 and DGCLOCK-V2 perform almost identically, could not be found.

1.4 Conclusion

The performance evaluation of the different page replacement algorithms showed that a low overhead per page hit and a fast selection of eviction candidates are as important for good performance as a high hit rate. In this context, it was demonstrated that the Timestamp-Sorting implementation of the LRU page replacement policy performed significantly better than the Hash-Map-Doubly-Linked-List implementation of the same page replacement policy due to its more scalable page reference statistic updating function on page hits.

The very low transaction throughput achieved with the relatively new LeanStore page eviction algorithm was surprising. Especially considering the fact that it was the only candidate in competition with enabled pointer swizzling in the buffer pool.

In most cases, the Timestamp-Sorting implementation of LRU may actually be recommended due to its consistently high transaction throughput. But due to concerns about the scalability of the sorting process for much larger buffer pools and more CPU cores, using ZCLOCK or a cleverly tuned DGCLOCK is probably safer.

1.4.1 Future Work

Future work should focus on eliminating the limitations mentioned in subsection 1.3.3. Other, more modern benchmarks such as *YSCB*[21] or *TPC-E*[22] should also be used and run on a more up-to-date server system. *GIS applications* also benefit from a well-optimized buffer pool—a comprehensive comparison of page replacement algorithms in this context would also be interesting.

The implementation of *many page replacement algorithms*, listed in section 1.2, was beyond the scope of this thesis—especially CAR [BM04], CART [BM04], CLOCK-Pro [JCZ05] and CLOCK-Pro+ [Li19] look very promising.

Hints from higher DBMS layers about the current and future page reference behavior together with *page prefetching* should also be combined in such a comprehensive evaluation.

[21] https://github.com/brianfrankcooper/YCSB/
[22] http://www.tpc.org/tpce/

2 Component-Wise Performance Evaluation of OLTP Systems

2.1 Harizopoulos et al. "OLTP through the Looking Glass, and What We Found There"

2.1.1 Introduction

As a starting point for the search for optimization potentials of OLTP systems, Harizopoulos et al. in [Har+08] have broken down the *CPU time* used by the *different components of a DBMS*. They stated that many of the features and guarantees provided by relational DBMS—which were mainly developed in the 1970's and 1980's—are not needed on *modern hardware* and for many *new applications*. Therefore, the CPU time consumed by the components responsible for realizing these features and guarantees can be seen as unnecessary overhead. Their work proves that significant performance improvements can be achieved with architectural changes for—what was called a few years later—NoSQL DBMS.

2.1.2 Features and Guarantees Identified to be Unnecessary

Depending on the application, one, some or all of the following features and guarantees typically provided by a relational DBMS have been identified as unnecessary by Harizopoulos et al.

Larger-Than-Memory Databases Since *database sizes grow more slowly than the available main memory*, it is no longer necessary to optimize a DBMS for DBs that are larger than the available main memory.

Multithreaded and Interleaved Execution If transactions never block due to disk access latency, *interleaved execution* using multithreading is no longer needed for good transaction throughput.

Strict Consistency and Transaction Support (and ACID in general) For many new OLTP applications—especially in the area of distributed Internet services—the *eventual consistency* will be sufficient. And a more complete approach—BASE (**B**asically **A**vailable, **S**oft state, **E**ventual consistency)—was later proposed by Pritchett in [Pri08].

Log-Based Recovery On the one hand, some OLTP applications do *not require recovery* at all, and on the other hand, recovery can be done from other sites of a *cluster*.

2.1.3 Affected DBMS Components

The authors propose the removal of components from a DBMS to eliminate these features and guarantees along with the associated overheads.

Buffer Management In-memory DBMSs simply allocate the DB in RAM.

Latching and Lock Management Without support for multithreading and interleaved transaction execution, latching and locking is not required anymore.

Log Management If crash recovery is required by the application, the required data can be provided by a *replicated DBS*. However, replication using *log-shipping* is not possible without a transaction log, so that other techniques must be used. The management of LSNs (log sequence numbers) is not necessary with a log-free DBMS.

B-Tree B-tree indices are mostly *optimized for disk-based use* [BM70] and other data structures—like ART [LKN13]—are better suited for in-memory DBMS. But using *larger pages* can still improve the in-memory performance of B-trees. But for the evaluation in [Har+08] the authors have simply *hand-optimized* the B-tree code for the case of *uncompressed integer keys* which are used throughout *TPC-C*.

2.1.4 Performance Evaluation

They have *analyzed these architectural changes quantitatively*, by taking the stock *Shore Storage Manager*[1][2] as a baseline and then *gradually removing* the

[1] https://research.cs.wisc.edu/shore/

[2] The *Shore Storage Manager* (the remaining part of the *Shore* DBMS) was derived from the *EXODUS*. *Shore* was later developed into *Shore-MT*, the direct predecessor of *Zero*,

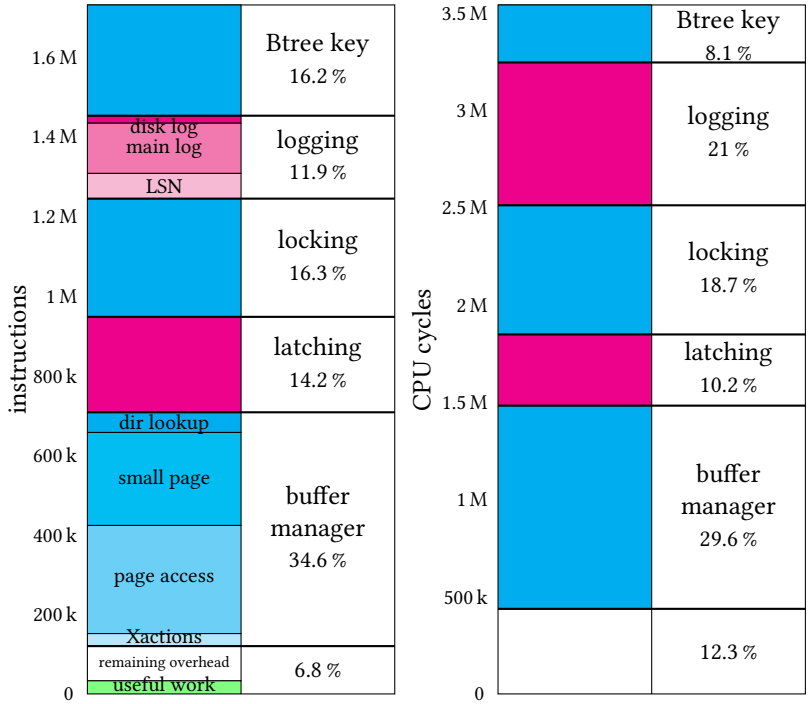


Figure 2.1: Number of CPU instructions and CPU cycles per *TPC-C* NEW ORDER transaction measured by Harizopoulos et al. for [Har+08]

previously mentioned components. This gave them for each of these DBMS components the *overhead* imposed on the overall system. They also applied some code optimizations to these components and to the remaining code, and tested the performance of a basic in-memory B-tree as a replacement for the DBMS. The used performance metrics are the *instruction count* and *CPU cycles* of the *TPC-C read-write transactions* NEW ORDER and PAYMENT.

Figure 2.1 shows their results for the NEW ORDER transaction. The number of CPU instructions executed during an average NEW ORDER transaction decreased by 16.2 % after optimizing the search for uncompressed integer keys in *B-tree* pages. Removing the I/O operations of the *log manager* (*disk log*), the generation of log records (*main log*) and the

which is used throughout this thesis.

maintenance of *LSN* reduced the number of executed CPU instructions by a further 11.9 % compared to the baseline. Removing calls to the *lock manager* and the *acquisition and release of latches* saved another 16.3 % respective 14.2 % of the instructions compared to the baseline. The authors achieved the greatest savings by removing parts of the *buffer manager*. They removed the directory used to find storage structures like indexes on disk (*dir lookup*), they increased the *page size* from 8 kB to 32 kB, they removed the facilities used to access disk pages and find disk pages in the buffer pool (*page access*) and they removed the transaction management (*Xactions*). Their bare B-tree in main memory executed only a tiny fraction of the CPU instructions (*useful work*) compared to the baseline.

Fewer instructions do not result in the same degree of fewer CPU cycles. The optimizations the authors applied to *Shore*'s *B-tree* did not remove many *CPU cache misses* and *branch miss predictions* compared to their optimizations to other DBMS components, resulting in lower savings in required CPU cycles. The comparatively small fraction of instructions saved after removing *transaction logging* resulted in a far greater proportion of CPU cycles saved because the logging code accessed *many memory locations*. The remaining components require around two CPU cycles per instruction.

2.1.5 Assessment of the Assumptions and Results

Larger-Than-Memory Databases The assumption that the DB of most OLTP applications fits in main memory is correct. But support for DBs larger-than-memory is still needed for *embedded DBs*—such as those used by web browsers for managing browser history, cookies, or bookmarks—and for *local copies* of production DBs on notebooks used to overcome network latency when working with e.g. GIS data in the home office.

Multithreaded and Interleaved Execution Without multithreaded and interleaved execution, even a modern single CPU socket desktop PC with *up to 64 CPU cores*[3] cannot be utilized to the slightest degree.

Strict Consistency and Transaction Support (and ACID in general) The idea that omitting transaction support and ACID guarantees is

[3] https://www.amd.com/en/products/ryzen-threadripper

beneficial for most modern web applications led to the rise of *NoSQL DBMSs in the 2000s*. These new systems were much more performant than traditional DBMSs and could be scaled horizontally. In the 2010s it was realized that the application development effort is higher when using a NoSQL DBMS and that relational DBMSs are possible with almost the performance and scalability of a NoSQL DBMS—*NewSQL DBMSs* were born. An overview of the transition from NoSQL to NewSQL and a definition of NewSQL DBMSs can be found in [PA16].

Performance Evaluation The *Shore Storage Manager* (as well as *Zero*, which is used in his thesis) *lacks the set-oriented layer* of a relational DBMS—it is therefore not fully representative for a traditional OLTP DBMS. But the greatest overhead of a relational DBMS should be below this layer, since it is mainly responsible for query optimization and compilation.

The similar architectures of traditional DBMSs allow such a performance evaluation to be *representative* to a certain extent, even if only one DBMS is examined. But evaluating multiple DBMSs would certainly have resulted in finding different optimization opportunities in these DBMSs, which would have led to different amounts of CPU instructions in the components.

In general, the *TPC-C* benchmark is a *suitable benchmark* for evaluating the OLTP performance of a DBS. However, the two *read/write transactions* selected by the authors (which account for 88 % of the *TPC-C* transactions) are not very representative for most web applications that use many *read-only transactions*. Most importantly, the performance gain from eliminating *transaction logging* is entirely based on the write accesses of these transactions. The standard transaction mixture of *TPC-C*—or for a focus on Web applications more ORDER STATUS and STOCK LEVEL transactions—should be used for the performance evaluation.

Harizopoulos et al. made it clear that they could not completely remove all components from *Shore*, but that by removing them in a certain order helped to minimize the overhead caused by these leftovers. However, if one wants to optimize a component instead of removing it completely, breaking down the CPU time into the different components is not very helpful because it is *too coarse*. A

different order of component removals would also lead to (slightly) different percentages of CPU time per DBMS component due to dependencies between different components. With the drawback of adding measurement overhead to the application under test—which slows it down and may distort the result—*profiling and tracing software* can be used to obtain a code line-, function-, sub-component-, or component-wise breakdown of CPU cycles or other performance metrics.

Revisited The measurements from [Har+08] are *repeated* in this chapter to address some of the problems of the original measurements. *Multithreaded and interleaved* execution and *all TPC-C transactions* are considered. And instead of removing DBMS components, *profiling and tracing* software on the baseline DBMS is used. The following section 2.2 describes the technique and gives an overview of the software that can be used for this and describes in more detail the software that was used. Subsection 2.3.3 gives results for single threaded DBS executions and subsection 2.3.4 discusses the multithreaded case. Section 2.4 presents some optimizations that can be easily applied based on these measurements.

2.2 Profiling and Tracing

The Oxford *A Dictionary of Computer Science* [BNK16] gives the following definitions:

> **Profiling** "Production of a histogram (or equivalent) concerning some aspect of a system. For example, an execution profile for a program might show the proportion of time spent in each individual procedure during a run of the program, while a statement profile might show the distribution of the statements in a program between the different kinds of statement provided by the language."
>
> **Trace Program** "A program that monitors the execution of some software system and provides information on the dynamic behaviour of that system in the form of a trace, i.e.

> a report of the sequence of actions carried out. Typically a trace program will offer several options as to the kind of trace produced. For example, there may be options to produce a statement-by-statement trace, or to trace just those statements that alter the flow of control, or to trace changes to the value of a specific variable."

A *program execution profile* is thus created by the aggregation of traces.

2.2.1 Software

While many *proprietary* performance analysis tools create execution profiles in one step—at least if one of the provided presets is suitable for the respective task—, the many *free and open-source* programs only collect program traces. Analysis of the collected traces and profiling can be done using scripts *written by the user* or *shared online*. But for common profiling tasks free and open source GUI frontends are available, at least for the more popular programs. Some trace programs are also *programmable*, which allows the user to perform more complex calculations during the tracing and to analyze the program execution in more detail, e.g. by reading variable values or function parameters, as is done in debugging.

This subsection contains a brief description and/or potential sample analyses of free and open source performance analysis tools that could have been used (partially) for the measurements for section 2.3. But for the simple analysis performed in section 2.3, the less powerful *Intel® VTune™ Profiler* was chosen because of its easy-to-use GUI and fast configuration. How to use it is described in more detail in subsection 2.2.2.

LTT The *Linux Trace Toolkit* is a *simple* and *not very configurable Linux* kernel tracer that logs system calls, traps, interrupts, memory management events, etc. The traces are then displayed using a graphical visualizer.

LTTng The *Linux Trace Toolkit Next Generation*[4] is a more sophisticated successor of *LTT*. It can trace the *Linux* kernel based on *built-in tracepoints* and *other instrumentation points*, but it can also be used to trace user

[4] https://lttng.org/

programs that need to be prepared by inserting tracepoints. Tracing is managed in isolated tracing sessions, where different event rules can be set per session for kernel and user space tracing. A trace record containing, for example, the process ID, call stack or CPU performance counter values is produced whenever an active event rule is satisfied. To analyze the generated traces, tools like *Babeltrace 2*[5] are useful.

ftrace *ftrace*[6] gives an insight into the operation of the *Linux* kernel. It requires a *Linux* kernel compiled with *ftrace* capabilities enabled, and is configured using virtual files—it is not a command line application. Events that are used to create traces are kernel *built-in static tracepoints*, *kprobe* events, *uprobe* events, or *any function call within the kernel*. Tracing of specific events can be enabled in the configuration. It can also be used to measure certain *latencies* within the kernel or to create a function graph. Tools for visualizing the not very human readable traces recorded by *ftrace* are of course also available.

perf *perf* is a performance analysis tool built into *Linux*, and it is probably the *most commonly* used one on *Linux*. It can create traces based on *hardware events* (e.g. for microarchitecture-level analysis), *low-level kernel events*, *kernel tracepoints*, *user-defined tracepoints* in user programs, dynamic *kprobe* or *uprobe* events, and a specific *trace frequency*. *perf* can create profiles that contain simple counter values, latencies aggregated per event, or human-readable aggregated call stacks with associated performance metrics.

DTrace *DTrace* is an open-source programmable tracing software that was originally released by *Sun Microsystems* for *Solaris* in 2005. Today, ports exist also for *macOS*, *Linux* and *Windows*.

Listing 2.1 shows a simple *DTrace* script written in the awk-like scripting language D, which is used by *DTrace*. vminfo:::pgpgin is an instrumentation point—a so-called probe—which fires upon the occurrence of a certain event in the virtual memory management of the operating system. According

[5] https://babeltrace.org/
[6] https://www.kernel.org/doc/Documentation/trace/ftrace.txt

```
#!/usr/sbin/dtrace -s

vminfo:::pgpgin {
      @pg[execname] = sum(arg0);
}
```

Listing 2.1: *DTrace* script pgpginbyproc.d from the *DTrace Toolkit*[7]

to the DTrace documentation[8] the pgpgin probe "fires whenever a page is paged in from the backing store or from a swap device". The number of pages paged in—returned by the probe in arg0—is then aggregated per process name (execname) into the aggregate pg using the sum() aggregation function. The aggregate pg thus contains for each process name (which possibly represents a number of running processes) that is paging in pages a counter of the number of pages paged in by it.

For more complex profiling tasks, many probes for kernel and user process tracing can be used in one script, resulting in *DTrace* scripts of 100s of lines. Brendan Gregg developed hundreds of scripts—covering many different fields—for his open source *DTrace Toolkit*, which is available at https://github.com/opendtrace/toolkit.

eBPF The *extended Berkeley Packet Filter* is an *in-kernel virtual machine* that can be used as a *programmable* open source tracing software similar to *DTrace*. It has its origin in the network traffic analysis tool *Berkeley Packet Filter* from 1992. While it runs small programs in the kernel whenever a certain event occurs, it is programmed using frameworks like *bpftrace*[9] or for more complex scripts *BCC*[10]—providing *high-level languages* like *DTrace*'s D—that run in user space. Like most profiling tools, *eBPF* supports the instrumentation points that are also supported by *perf*.

Listing 2.2 shows a simple *bpftrace* script written in an awk-like language, which is compiled to *eBPF* bytecode and then executed by *eBPF* inside the kernel. The action block of the special BEGIN event is executed once when the script is started. And every time an event matching

[8] http://dtrace.org/guide
[9] https://github.com/iovisor/bpftrace
[10] https://github.com/iovisor/bcc

```
#!/usr/bin/env bpftrace

BEGIN {
    printf("%-10s %-5s %s\n", "TIME(ms)", "PID", "ARGS");
}

tracepoint:syscalls:sys_enter_exec* {
    printf("%-10u %-5d ", elapsed / 1e6, pid);
    join(args->argv);
}
```

Listing 2.2: *bpftrace* script execsnoop.bt, which is one of the official examples of *bpftrace*

tracepoint:syscalls: sys_enter_exec* occurs, where * is a wildcard for a $0{-}\infty$ number of characters, the elapsed time (since script start) in ms, the corresponding process ID, and the array of event arguments args->argv with a space as delimiter are printed to the standard output stream. The wildcarded event matches e.g. the tracepoint tracepoint:syscalls:sys_enter_execve, which is fired whenever a new process is started, so this script prints information about each new process.

ktap *ktap*[11] is a free and open-source, but *discontinued in-kernel Lua virtual machine* for *Linux* and works almost exactly like *eBPF*.

SystemTap *SystemTap*[12] is, unlike *eBPF* or *ktap*, not a kernel-internal virtual machine, but a program that *compiles user-written programs into kernel modules* and executes them dynamically in the *Linux* kernel. This makes it very powerful for instrumentation of live running systems and keeps the overhead very low. The scripting language used is hardly distinguishable from those of *DTrace*, *bpftrace* (*eBPF*) or *ktap*—the general script structure with *probe points* (pattern matching for events) and associated code blocks with C-style syntax is inspired by the script language *awk*.

Since the scripts can manipulate the entire system, *SystemTap* checks them before adding them as a kernel module by running them a few times

[11] https://github.com/ktap/ktap
[12] https://sourceware.org/systemtap/wiki

```
#! /usr/bin/env stap

/*
 * Copyright (C) 2006–2007 Red Hat Inc.
 *
 * This copyrighted material is made available to anyone wishing to use,
 * modify, copy, or redistribute it subject to the terms and conditions
 * of the GNU General Public License v.2.
 */
global start

function timestamp:long() {
        return gettimeofday_us() - start
}

function proc:string() {
        return sprintf("%d (%s)", pid(), execname())
}

probe begin {
        start = gettimeofday_us()
}

probe syscall.nanosleep.return,
syscall.compat_nanosleep.return ?
{
        elapsed_time = gettimeofday_us() - @entry(gettimeofday_us())
        printf("%d %s %s: %d\n", timestamp(), proc(), name, elapsed_time)
}
```

Listing 2.3: *SystemTap* script sleeptime.stp, which is one of the official examples of *SystemTap*

outside the kernel to minimize the risk of *kernel panics* and other serious errors. However, it is also possible to add any C code to *SystemTap* scripts, which can be quite harmful.

Listing 2.3 is a simple *SystemTap* script which logs calls to the system calls nanosleep and compat_nanosleep and the respective time periods. The special probe point begin writes the current epoch time in µs to the global

variable start when the script is started. The function timestamp returns the µs since the script was started as long value. The proc function returns a formatted string containing the process ID and process name of the process that triggered the probe from which the function was called.

The probe defined at the end of the script is always executed when the system call nanosleep or compat_nanosleep returns. From inside a return probe, @entry can be used to execute code at the time of the corresponding function call. Thus, elapsed_time contains the time between the call and the return of a nanosleep or compat_nanosleep system call. The probe outputs the µs since the script was started, information about the process that caused the probe to execute, the name of the system call (either nanosleep or compat_nanosleep) and the elapsed_time to the standard output stream.

Valgrind *Valgrind*[13] was originally a free and open-source memory debugger for *Linux* but evolved into a more generic profiling tool for *Linux*, *macOS* and *Solaris*. It is different from all the other programs presented here as it takes the program to analyse in the form of a x86 binary as input and runs it on a *synthetic CPU* where *Valgrind* tools do e.g. memory error detection or cache and branch prediction profiling of that program.

2.2.2 Intel® VTune™ Profiler

Intel® VTune™ Profiler[14] is a proprietary profiling software that is available as a *stand-alone product* or as a part of *Intel® Parallel Studio XE*[15] or *Intel® System Studio*[16]. It is available for *Windows*, *macOS*[17] and *Linux*, but it also supports target systems for analysis on which some other operating systems like Android run.

In contrast to most open-source solutions for tracing and profiling, the *Intel® VTune™ Profiler* offers a quite *polished GUI* that provides a *quick entry* into software profiling. However, to automate the software analysis with this tool, the collector—the part of the software responsible for data

[13] https://www.valgrind.org/
[14] https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html
[15] https://software.intel.com/content/www/us/en/develop/tools/parallel-studio-xe.html
[16] https://software.intel.com/content/www/us/en/develop/tools/system-studio.html
[17] There is no collector available for *macOS*, which prevents it from being a target system.

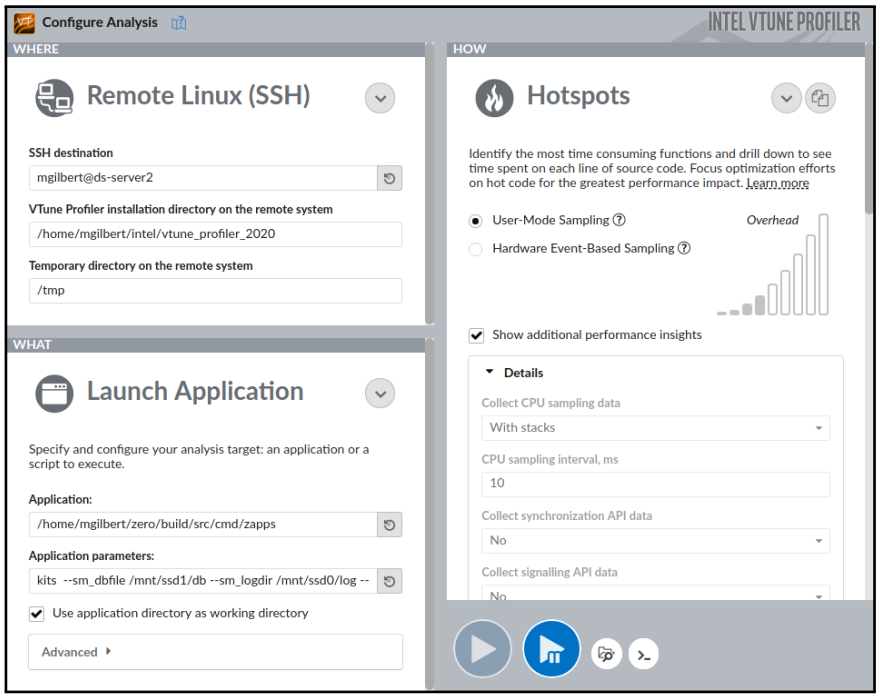


Figure 2.2: In *Intel® VTune™ Profiler*, the configuration of an analysis via GUI is clearly structured: **How** should **what** software running **where** be analyzed?

acquisition and post-processing—can also be used via the command line. The configuration of an analysis via the GUI even allows the export of the respective command to make scripting even easier. But any more complex software profile should be viewed in the GUI.

The software allows naming of the generated analyses and their management in *projects*, which makes it easier to keep track of all generated results. But these functionalities are not as comfortable as they could be—an analysis cannot be named during configuration. The results have to be generated first to give it its own name afterwards. And after renaming an analysis, the results must be reloaded by the software—resetting the analysis viewer.

But before the first analysis can be created, a *license* for the software must

be acquired and the *VTune™ Profiler* needs to be installed. The *installation* can be done via command line or GUI. But for example, if the target software is running on a server, only the *sampling driver* needs to be installed there, while it is controlled by a full *VTune™ Profiler* installation on the PC on which the analyses are initiated and evaluated.

But once the software (and the sampling driver on a potentially separate target system) is installed, the first analysis can be started. Figure 2.2 shows the *configuration dialog* for a *VTune™ Profiler* analysis via GUI. It is divided into *three main topics*.

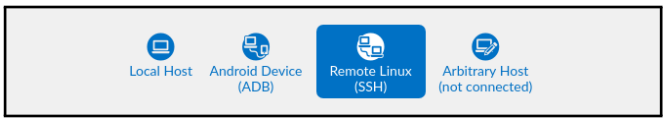


Figure 2.3: The *Intel® VTune™ Profiler* does not only allow the analysis of software running local.

First, it is required to decide **where** to run the target software on—figure 2.2 shows the different options. For example, if "Remote Linux (SSH)" is selected, as in the example shown in figure 2.2, the local *VTune™ Profiler* will connect to the target system, where at least the *sampling driver* must be installed, via *SSH*, conveniently using the user's configuration for SSH connections.

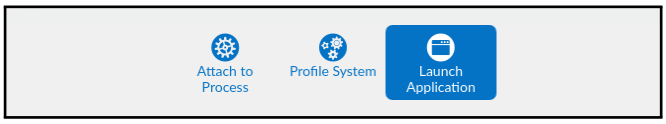


Figure 2.4: The *Intel® VTune™ Profiler* can also attach to an already running process to analyze that.

The second decision to be made is the question of **what** to analyze on the target system by the *VTune™ Profiler*. Figure 2.4 shows the different options. While "Profile System" and "Attach to Process" can also be used to evaluate a production system, the "Launch Application" option is best suited for the measurements performed for this chapter. With this option, the target application is started by the *VTune™ Profiler* and specified parameters are passed to it. Data acquisition can be delayed, and a maximum amount of

data or time span for data acquisition can be specified, for example, to omit the start and shutdown procedures of the target application from analysis.

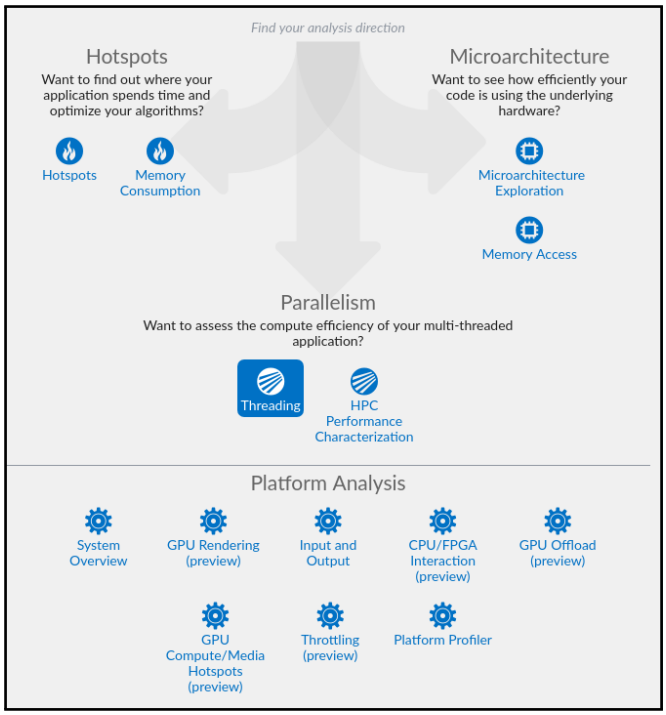


Figure 2.5: The *Intel® VTune™ Profiler* offers a number of presets that define which events should be collected during analysis and how this data should be post-processed afterwards.

With the final decision, the user tells the *VTune™ Profiler* their intention for the analysis—**how** should the *VTune™ Profiler* create the analysis? The *Intel® VTune™ Profiler* provides a set of *presets* for various analyses—each preset defines the *events to be counted* and the *sample rate* for *stack traces*, allowing the identification of components, functions, or code lines causing these events.

The preset can be used for example with "User-Mode Sampling" or with "Hardware Event-Based Sampling". "User-Mode Sampling" *interrupts* the analyzed process (defined in "what") every 10 ms (default "CPU sampling interval") and stores the current *call stack* of each unhalted thread

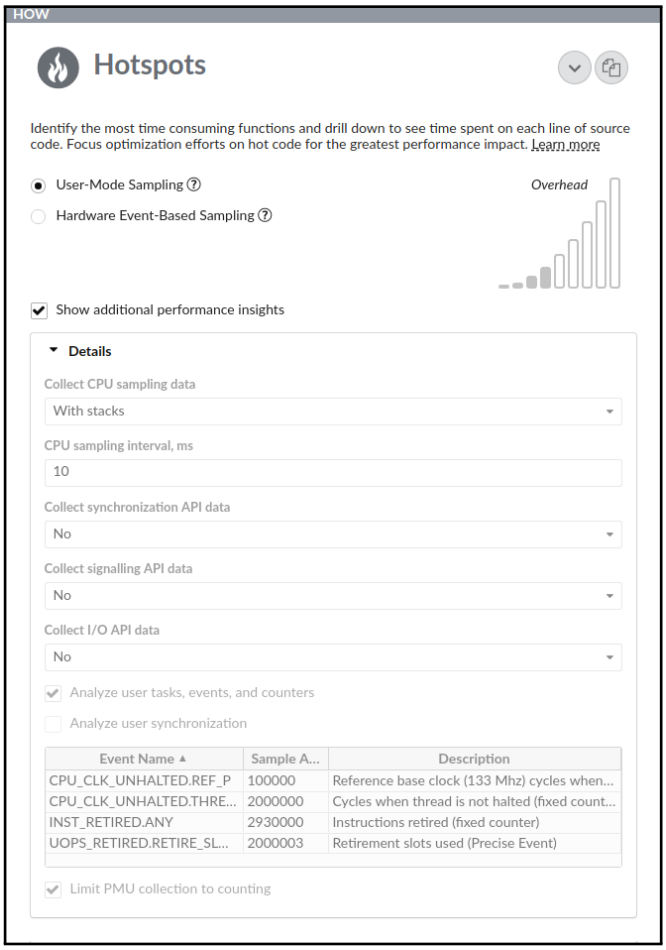


Figure 2.6: The "Hotspots" preset of the *Intel® VTune™ Profiler* specifies among many other options the CPU sampling interval and the events to be counted. Many less relevant options that are not used by this preset have been removed from this screenshot.

of the process. The CPU time since the previous sample is then assigned to that captured call stack. Afterwards, the stack trace is *aggregated* by summing the CPU times of identical call stacks to obtain a—statistically exact—allocation of CPU time to different call stacks. "Hardware Event-

Based Sampling" also interrupts the analyzed process to store the call stacks, but it assigns the values of certain *hardware counters* to each sample, so that different samples contribute to the final summary based on these values. These hardware counters can count e.g. CPU cycles, executed instructions, L2 cache misses or branch mispredictions—depending on the capabilities of the CPU architecture. Figure 2.6 shows some settings of the "Hotspots" preset, but many others, e.g. regarding GPU usage and memory, have been removed from the screenshot for the sake of clarity.

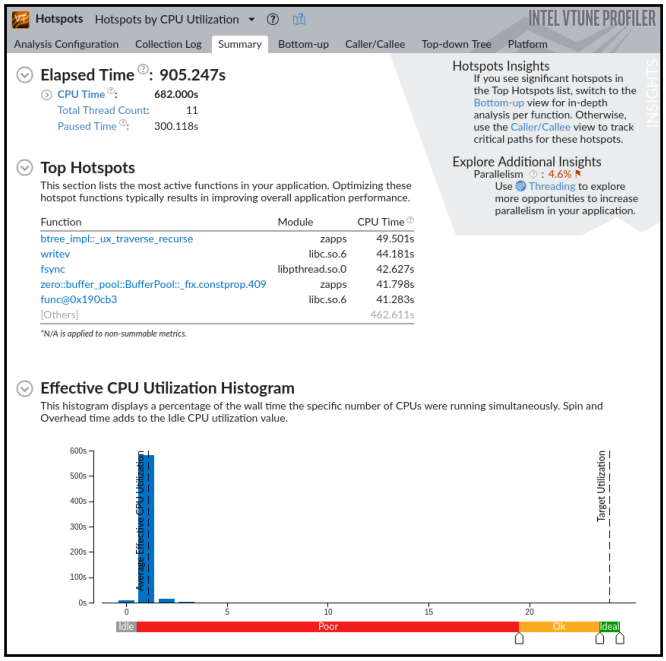


Figure 2.7: The *Intel® VTune™ Profiler* provides a "Summary" tab for each analysis performed, displaying—depending on configuration—highlights related to various performance aspects.

The *VTune™ Profiler* presents the results of each analysis in a variety of ways, divided among different tabs. An exemplary "Summary" tab of the "Hotspots" preset when using "User-Mode Sampling" is shown in figure 2.7. It shows a *ranking of the functions* that required the most CPU time and a visualization of the *degree of parallelism* achieved.

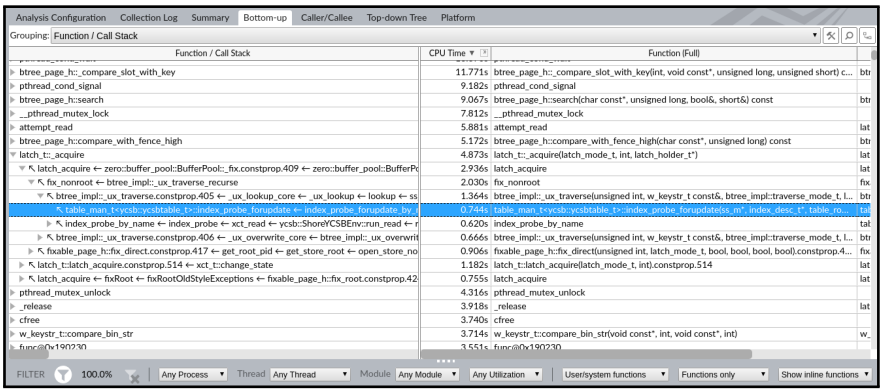


Figure 2.8: The "Bottom-up" tab of a *Intel® VTune™ Profiler* analysis summarizes the collected samples based on the lowest element of the call stacks.

The "Bottom-up" tab breaks down CPU time (or other performance characteristics) based on the *bottom element of the call stack* of each sample. This element is the function (or loop, if displayed) that was running on the CPU at the time the sample was taken. As shown in the figure 2.8, the callers (and their callers) to these functions can also be displayed along with a breakdown of the share of CPU time of each combination.

The samples considered for the summaries in the "Bottom-up" tab and the other tabs can be *filtered* by timestamp, process, thread and software module. These filter options are shown at the bottom of the figure 2.8.

The "Caller/Callee" tab lists *each function contained in the stack trace* and assigns it the CPU time it requires including that of any functions it calls. Therefore, the entry point of the program is usually the function with the highest CPU time required—it is the base of every call and every thread spawn. This list is shown in the *left frame* in figure 2.9. The callers and callees to these functions can then be retrieved as shown for fix_nonroot in the example. The *upper right frame* in the figure shows the callers as they are displayed in the "Bottom-up" tab. The callees can be displayed in the same way as they are shown in the "Top-down Tree" tab. This is shown in the *lower right frame* in figure 2.9.

The "Top-down Tree" tab initially shows only the entry point of the analyzed program and the total CPU time it has used during the analysis.

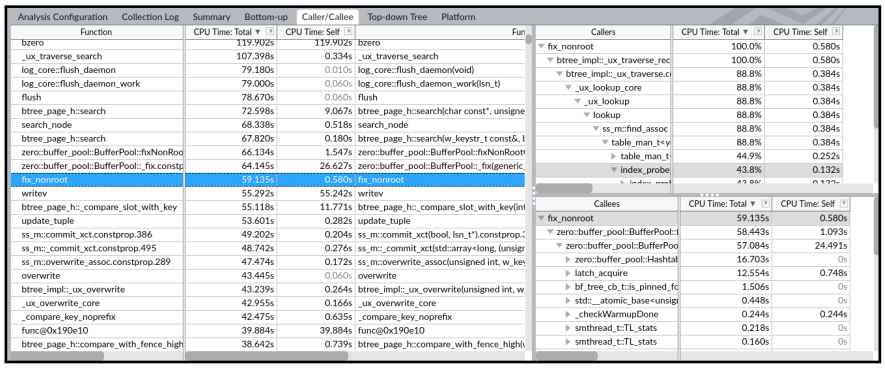


Figure 2.9: The "Caller/Callee" tab of a *Intel® VTune™ Profiler* analysis lists the total CPU time used by each function—including the called functions.

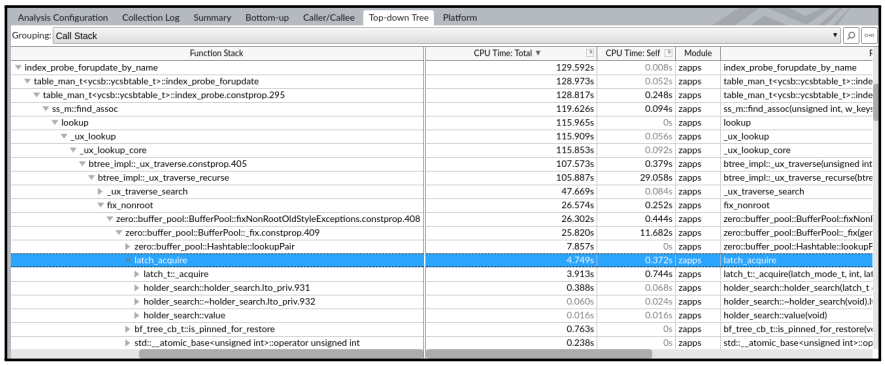


Figure 2.10: The "Top-down Tree" tab of a *Intel® VTune™ Profiler* analysis offers the possibility to interactively discover the call stack of a program from top to bottom, sorted by e.g. CPU time.

From there, the view can be *expanded*, as shown in figure 2.10, to show all called functions (and the functions they call, etc.) together with the CPU time they utilize (including or excluding their callees).

Finally, the "Platform" tab visualizes the complete analysis in terms of *CPU usage*. As shown in figure 2.11, it shows a graph for each thread (or alternatively process, module, or any combination of them), with CPU time displayed in brown and idle time in green. For example, if "synchronization

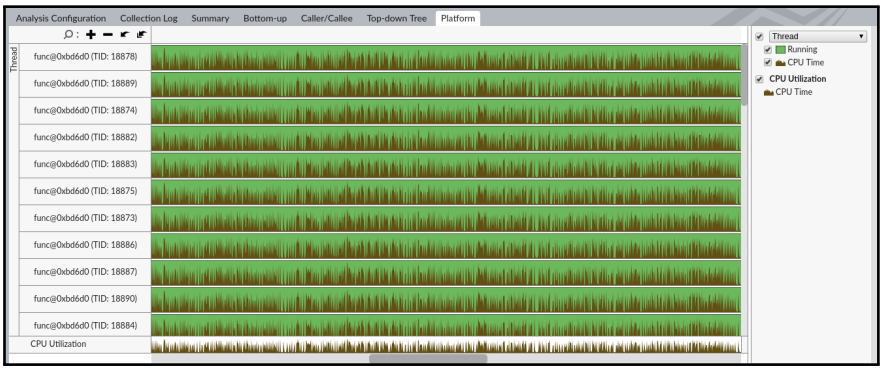


Figure 2.11: The "Platform" tab of a *Intel® VTune™ Profiler* analysis visualizes e.g. the CPU utilization (and optionally spin time of spinlocks) per thread, process or module.

API data" is collected, the spin time of spinlocks would be displayed in the graphs in red.

2.2.2.1 Flaws

While collecting measurements for the section 2.3, the *Intel® VTune™ Profiler* showed some of its flaws.

When "Launch Application" is used, it starts the target application and shows the command line output of this program during the tracing. However, this command line output is not saved together with the trace data and thus discarded when the analysis results are closed. If anything from the command line output of the analyzed program is to be used later with the analysis results—such as transaction throughput in the case of *Zero*—this information must be stored outside the *VTune™ Profiler*.

2.3 Measurements of Harizopoulos et al. – Redone

Based on the estimation that the assumptions of Harizopoulos et al. and their methodology are too restrictive, in this section their measurements are repeated with *changed assumptions* and using a *completely different technique*. Subsection 2.3.1 describes in detail the methodology used and

to demonstrate that the—-probably most controversial—assumption from [Har+08]—-the unnecessity for multithreaded execution of OLTP DBS—is invalid, subsection 2.3.3 shows the new evaluation (mostly) single-threaded while subsection 2.3.4 uses multithreaded execution to provide data for a comparison.

Due to the extensive evaluation of the effect of the buffer pool size on OLTP performance in Chapter 1 and the fact that today larger-than-memory databases are rarely used, *only in-memory DBSs* are considered in these measurements. Since *transaction support* and *ACID guarantees* are still used in most OLTP applications, these are also used during the measurements. *ARIES-style logging* is also used, as this is still common and built into the used prototype DBMS.

2.3.1 Methodology

The software used to breakdown the runtime of an OLTP system is the *Intel® VTune™ Profiler 2020*, which is described in subsection 2.2.2. It was configured according to the outlined configuration steps as follows:

Where The DBS was running on a "Remote Linux (SSH)" server. Its configuration is described in subsection 1.3.1.

What The DBS with built-in benchmarks was started by the profiling software. Some details about the embedded DBMS used—*Zero*—were given in the subsection 1.3.3 and the benchmarks are *TPC-C* (see subsection 1.3.2) and *YCSB* (see subsection 2.3.1.1). The DBS started with a clean DB (no crash recovery required), but—obviously—with an empty buffer pool. Due to the fact that a typical DBMS runs continuously—and thus with a populated buffer pool—the *Intel® VTune™ Profiler* started tracing after a cold start of 5 min (recording 10 min total) to give realistic results.

How The preset "Hotspots" of the *Intel® VTune™ Profiler* was used.

The software profile generated by the *Intel® VTune™ Profiler* consists of stack traces, as shown in subsection 2.2.2, with a total run time in seconds for each of them. Every relevant (except e.g. benchmark code such as the test data generation that would typically run on a client) stack trace is then *manually assigned to one of the DBMS components* described in subsection 2.3.1.2. The delayed start of tracing and the overhead associated with it

prevent a reliable calculation of the CPU time per transaction, so that only the percentage of CPU time used per component is shown.

2.3.1.1 Benchmarks

The first benchmark used for the measurements in this chapter is the *Yahoo! Cloud Serving Benchmark*[18] (*YCSB*). It is based on a simple workload for key-value stores (*NoSQL DBMSs*) and was designed for systems running in the cloud. It is for NoSQL DBMSs what *TPC-C* is for OLTP SQL DBMSs.

The DB contains 10 000 000 completely random *key-value pairs* in a table with 10 character fields of 100 B each mapped to a 10 B key. One *skewed* (based on the Zipfian distribution) random key is drawn per transaction. A read transaction reads all 10 fields associated with the key and an update transaction writes random data to a randomly selected field associated with the key. The read-only *YCSB* workloads from subsection 2.3.3.1 and 2.3.4.1 execute only read transactions, the write-only ones from subsection 2.3.3.2 and 2.3.4.2 execute only update transactions. No transactions abort and all queried keys exist in the DB.

The second benchmark used in this chapter—*TPC-C*—performs read and (more frequently) update transactions, so that a read-write *YCSB* workload is evaluated. The database and transactions of *TPC-C* have already been described in the subsection 1.3.2.

2.3.1.2 Considered DBMS Components

B-Tree Each stack trace associated with maintaining the *Foster B-Tree* structure or searching for keys within a B-Tree page is assigned to this component.

Logging The CPU time used to create log records, maintain log sequence numbers (LSNs) and flush log records to SSD is assigned to this component.

Locking The CPU time spent on *transaction management*, acquiring and releasing locks and managing locks in the *lock manager* is assigned to this component.

[18] https://ycsb.site/

Latching *Internal structures* of transaction and lock managers require latches (not the mutexes on which the locks are based on) for synchronization in multithreaded environments. The CPU time required to acquire and release these latches is assigned to this sub-component.

Buffer Pool The effort for *fixing and unfixing pages* is assigned to this component. In addition to the following sub-components, page eviction would also be a sub-component, but only the RANDOM page replacement algorithm and smaller-than-memory DBs are considered thus requiring no CPU time for page eviction.

Latching A latch is assigned to *each buffer frame*, and a thread reading or writing a page must acquire the corresponding latch in the appropriate mode (shared or exclusive). The CPU time required to acquire and release these latches is assigned to this sub-component.

Fetching Few *pages are fetched from the SSD* while the stack traces are being gathered, because—due to the skewed access to the records—not all pages were accessed during the cold start phase, which is ignored by the profiler. But since the whole database fits into the buffer pool during each benchmark run, no pages need to be fetched because they were previously evicted. The CPU time required to fetch pages from the SSD is assigned to this sub-component.

Hash Table The hash table *maps the page identifier of each page in the buffer pool to the corresponding buffer frame index*. It is queried on each page fix—regardless of whether it is a page hit or a page miss. The runtime required to search within this hash table is assigned to this sub-component. More details about the hash table and an alternative can be found in the subsection 2.4.1.

Free List The free list is a simple concurrent *queue* containing the *indexes of unoccupied buffer frames*. Indexes are only dequeued when a page is fetched from SSD, so the CPU time assigned to this sub-component is very little.

The *logical access paths* layer, which is responsible for the mapping between internal and external records, among other things, and which is

not a core component of *Zero*, is not considered here. It consumes up to 25 % of the CPU time, but mainly for memory operations, e.g. for creating records based on input data, so these operations would be considered *useful work* by Harizopoulos et al.

2.3.2 Other Related Work

In contrast to Harizopoulos et al., Johnson et al. focus in [Joh+09] on scalability in the context of multithreading. They present their successor of the *Shore Storage Manager*—*Shore-MT*—together with measurements to prove its better multithreading capabilities. Instead of removing modules of the *Shore Storage Manager* one by one, they *optimized them one after the other* and showed the increase in transaction throughput (for a simple insert-only microbenchmark) after each step. In this way, their measurements show which components of *Shore Storage Manager* limit scalability to what extent.

Pandis et al. proposed a new page latching protocol for partitioned DBSs. In their work, they compared their approach—physiological partitioning (PLP)—to the "conventional" approach, by breaking down the *number of critical sections* per DBMS component.

The most detailed analysis of the inner workings of a DBMS in terms of performance was conducted by Tözün et al. in [Töz+13]. Their main goal was to compare the (micro-architectural) behavior of the OLTP benchmarks of the *Transaction Processing Performance Council*—*TPC-B*, *TPC-C* and *TPC-E*. They broke down *CPU time* and *other performance metrics per DBMS component* in a very similar way as here, compared CPU cache misses per cache level, and compared the behavior of the three benchmarks on systems that use *Hyper-Threading Technology*.

2.3.3 Single-Threaded OLTP System Analysis

The analysis of single-threaded OLTP systems is especially relevant for the comparisons between the results of Harizopoulos et al. and the results presented here, which were obtained using a totally different methodology. However, the results can also give a better insight—compared to the results for multithreaded OLTP systems from subsection 2.3.4—into the *overhead*

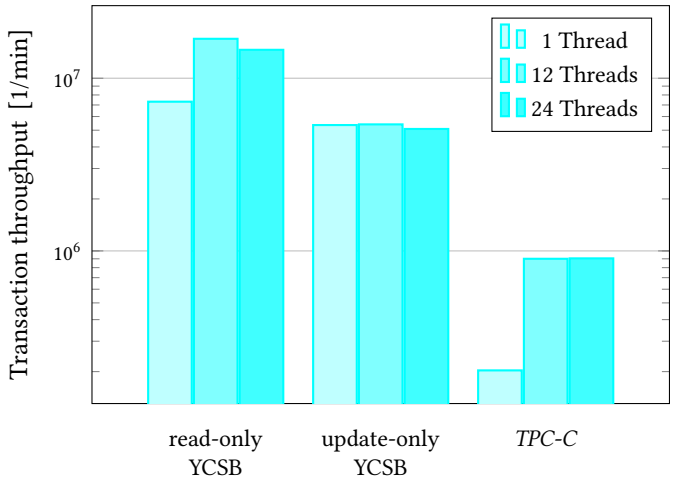


Figure 2.12: Transaction throughput for different numbers of working threads for read/update-only *YCSB* and *TPC-C*

of the pessimistic concurrency control, because here—without multithreaded and interleaved execution—every operation performed by the lock manager is unnecessary.

A quick comparison of the transaction throughput achieved with *different numbers of working threads*—i.e., different numbers of threads processing transactions *concurrently* and different numbers of transactions executing in an *interleaved* fashion—is shown in figure 2.12. The used server (see subsection 1.3.1) is equipped with two CPUs, each with 6 cores, supporting 2 hardware contexts running concurrently via simultaneous multithreading, so that a total of 24 *logical cores* or 12 *physical cores* are available.

The results indicate that the *additional logical cores* cannot be used to increase transaction throughput. In fact, they even increase overhead through more contention and additional context switches. But *read-only YCSB* and especially *TPC-C* benefit from multithreaded and interleaved transaction execution. *Read-only YCSB* can more than double its throughput if 12 instead of 1 working threads are used, *TPC-C* can more than quadruple its throughput. The bottleneck of *read-only YCSB* is the contention caused by the very large number of very short transactions on a global latch in the transaction manager, which has to be acquired twice per transaction. The

limited write rate of the transaction log prevents an increase in transaction throughput for *update-only YCSB*. The more balanced workload of *TPC-C* can better utilize the larger number of working threads.

2.3.3.1 Read-Only YCSB

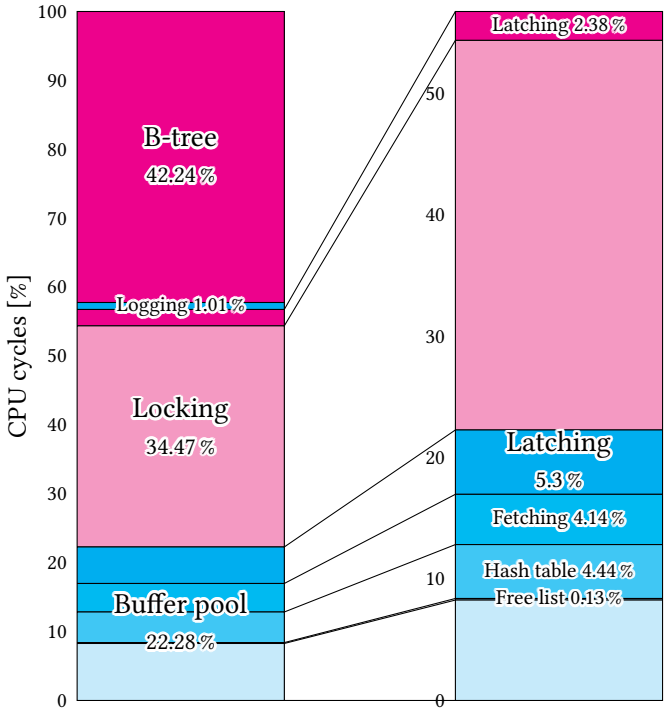


Figure 2.13: Component-wise breakdown of the CPU cycles when running read-only *YCSB* with one querying terminal.

Figure 2.13 shows the breakdown of CPU time of *Zero* when running *read-only YCSB* on 1 working thread and a DB smaller than the buffer pool. While this diagram only shows the percentage of CPU time per DBMS component, CPU times in seconds are given in table 2.1 later in this section.

Since *read-only* workloads do not generate log entries, *logging* here only accounts for 1.01 % of the total CPU time. The only overhead of the transaction log is caused by the need to store the current LSN for each

transaction at the time it begins.

The 4.14 % of CPU time used to *fetch pages* from the SSD (and the 0.13 % used to access the buffer pool's free list) is *caused by the skewed DB accesses* of *YCSB* where not all pages used during the benchmark run were referenced during the buffer pool warmup.

Because of the very large number of simple transactions that are being executed, the lock manager and mainly the *transaction manager* use 34.47 % of CPU time. This overhead is caused by management tasks required whenever a transaction begins or commits.

During single-threaded execution, no concurrency control is required, so *lock management* and *buffer frame latching*—which accounts for 5.3 % of CPU time—are a waste of resources.

2.3.3.2 Write-Only YCSB

Figure 2.14 shows the CPU time breakdown of *Zero* when running *update-only YCSB* on 1 working thread and a DB smaller than the buffer pool.

Just like the *read-only YCSB*, the *update-only YCSB* does not access all used DB pages during buffer pool warmup. However, due to the lower transaction throughput of this workload, significantly fewer pages have to be *fetched* using only 1.17 % + 0.07 % of the total CPU time.

In contrast to *read-only* workloads, each transaction of the *update-only YCSB* results in the creation and writing of a log entry, which results in 24.42 % of the CPU time used for *logging*.

The just a little smaller number of simple transactions executed with the *update-only* variant causes the *lock and transaction managers* to use 21.73 % of CPU time. As can be seen in table 2.1, the absolute CPU time used by these components reflects what is expected taking into account the transaction throughput and CPU time required during read-only *YCSB*.

The lower transaction throughput does not reduce the absolute CPU time used by the *Foster B-tree*, although fewer records are accessed. The lower overhead imposed by the *buffer pool*—compared to the measurement with *read-only YCSB*—is similar to what would be expected considering the lower transaction throughput.

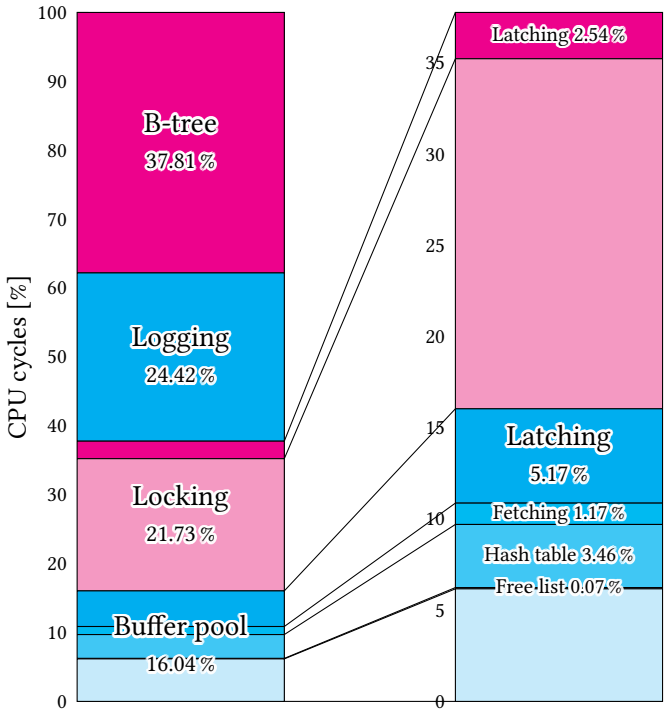


Figure 2.14: Component-wise breakdown of the CPU cycles when running update-only *YCSB* with one querying terminal.

2.3.3.3 TPC-C

Figure 2.15 shows the breakdown of CPU time of *Zero* when running *TPC-C* with one working thread and a DB not growing beyond the size of the buffer pool.

The *TPC-C* benchmark is not much less write-intensive than *update-only YCSB*. In addition to the updates performed by *YCSB*, *TPC-C* also inserts many records, resulting in a much larger *logging* overhead of 42.46 % of the total CPU time. This is twice as high a share as that measured by Harizopoulos et al. for the NEW ORDER transaction. This means that either the PAYMENT transaction—the other update transaction executed here—causes much more *logging* overhead or that the log manager of *Zero* is much less optimized compared to the other components of the DBMS.

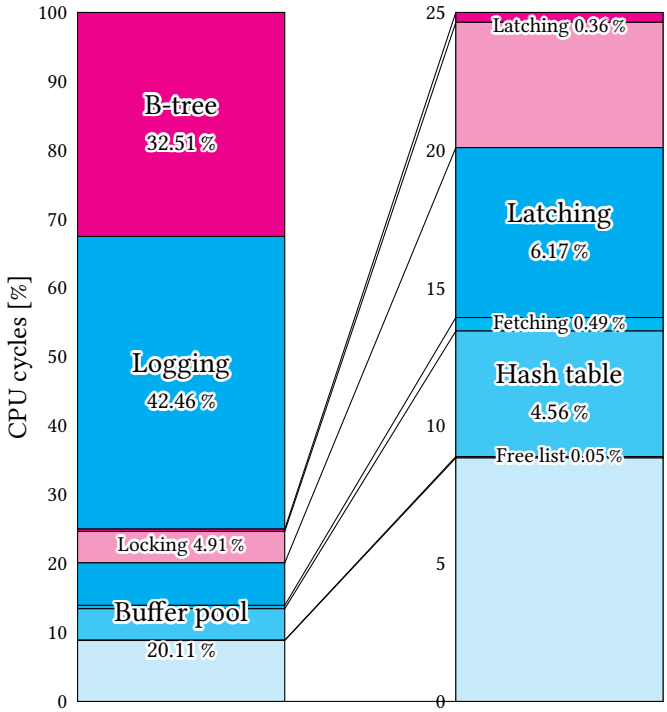


Figure 2.15: Component-wise breakdown of the CPU cycles when running *TPC-C* with one querying terminal.

The fewer transactions that are executed, the less CPU time is used by the transaction management. Correspondingly, overhead of *lock and transaction management* is just at 4.91 %.

TPC-C fixes more pages—compared to *YCSB*—containing records from several smaller *B-trees* which results in fewer executed B-tree instructions (32.51 % of CPU time) and more *buffer pool* work (20.11 % of CPU time).

Harizopoulos et al. measured a much higher *concurrency control* overhead and also a higher *buffer pool* overhead, but much less overhead in the *B-tree*. The different results for the B-tree can be explained mainly through the different approaches—the total CPU time used by the B-tree is given here, while they measured only the benefit obtained with a small optimization. But especially the *lock management and latching* seems to be more optimized in *Zero* compared to its predecessor, the *Shore Storage Manager*.

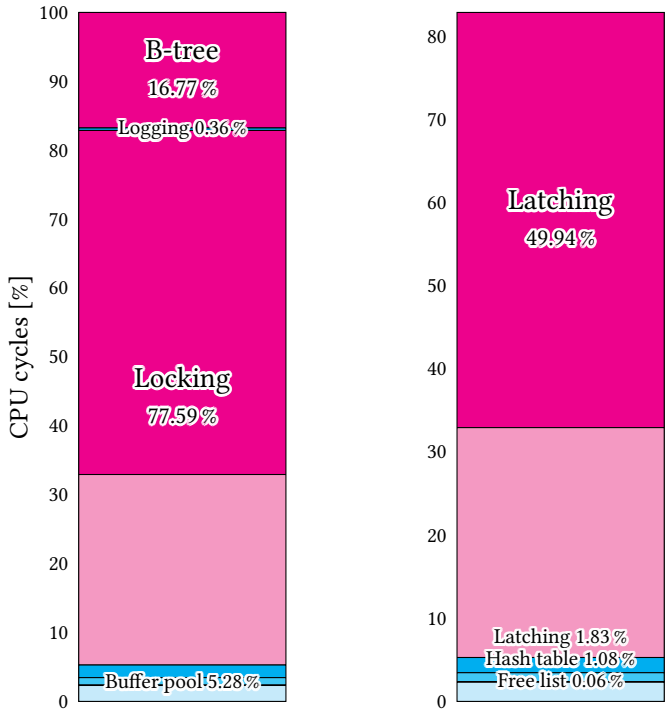


Figure 2.16: Component-wise breakdown of the CPU cycles when running read-only *YCSB* with 24 querying terminal.

2.3.4 Multithreaded OLTP System Analysis

Based on the trend of having *more and more cores per CPU* and the performance benefits of multithreaded execution for OLTP workloads (*TPC-C*) as shown in figure 2.12, these results for a system with 24 working threads should be considered *more relevant* to modern OLTP systems than the results in the previous section. However, the traditional *two-phase locking* (2PL) used by *Zero* is less representative because almost all modern (main memory) DBMSs use *multiversion concurrency control* (MVCC)[19], which is considered more scalable with the number of threads when correctly designed [Wu+17]. The 10 years old CPUs used for this analysis do not represent the state of the art in terms of the degree of achievable parallelism.

[19] MVCC was originally proposed by Reed in [Ree78].

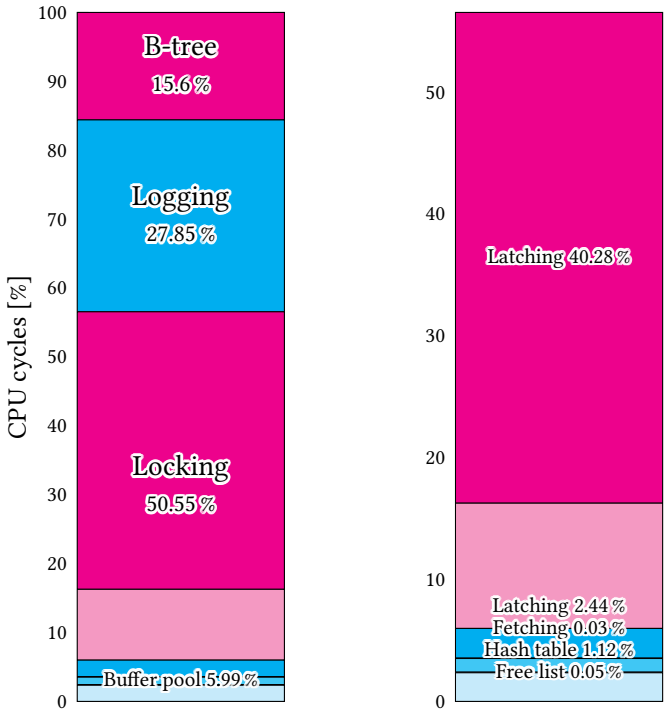


Figure 2.17: Component-wise breakdown of the CPU cycles when running update-only *YCSB* with 24 querying terminal.

2.3.4.1 Read-Only YCSB

Figure 2.16 shows the CPU time breakdown of *Zero* when running *read-only YCSB* with 24 working thread and a DB smaller than the buffer pool.

The high percentage of CPU time used by the *lock and transaction managers* is much more significant here—it dominates every other component—compared to single-threaded execution. The 77.59 % of the total computation time are mostly used spinning a *global latch* inside the transaction manager implemented using a spinlock (almost 50 % of CPU time is used for this).

Due to the dominance of the transaction manager with regard to used CPU time, the overhead of *logging* is here—where no log records are created—down to 0.36 % of the total CPU time.

As can read from table 2.1, the absolute CPU time used by the *Foster B-tree* and *buffer pool* grew more strongly than the transaction throughput compared to the single-threaded case. The increase in the buffer pool can partly be explained by a higher latching cost.

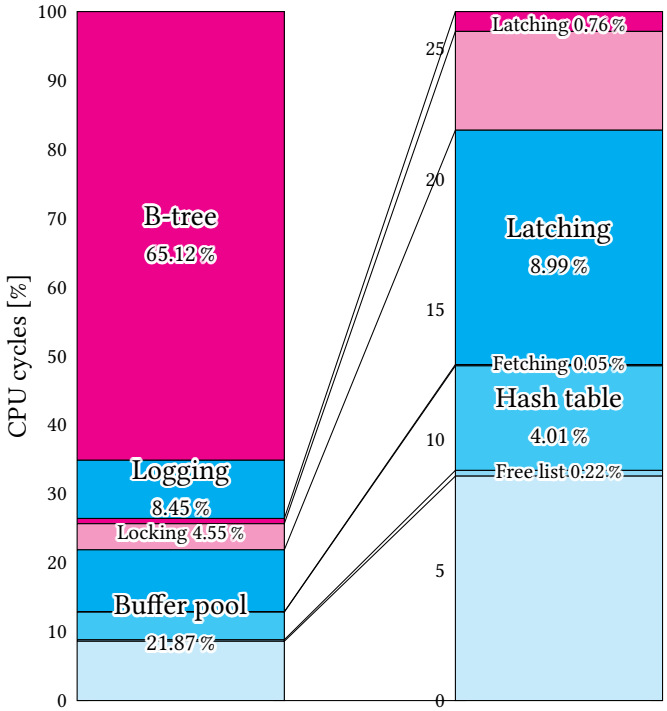


Figure 2.18: Component-wise breakdown of the CPU cycles when running *TPC-C* with 24 querying terminal.

2.3.4.2 Write-Only YCSB

Figure 2.17 shows the CPU time breakdown of *Zero* when running *update-only YCSB* with 24 working thread and a DB smaller than the buffer pool.

The same spinlock that causes the high overhead of the *lock and transaction managers* for *read-only YCSB* also causes a high overhead of these components here. However, due to the lower transaction throughput, the CPU time is not quite as high.

	1 thread			24 threads		
	read-only YCSB	update-only YCSB	TPC-C	read-only YCSB	update-only YCSB	TPC-C
logging	5.236	166.306	244.855	17.566	1119.645	316.533
locking	166.703	130.689	26.257	1367.299	413.172	142.108
+ latching	12.374	17.269	2.08	2470.092	1619.046	28.26
B-tree	219.454	257.482	187.497	829.531	627.305	2437.772
buffer pool	42.984	41.989	50.993	114.462	94.723	322.127
+ latching	27.562	35.204	35.584	90.462	98.321	336.345
+ hash table	23.053	23.582	26.319	53.211	44.874	150.108
+ free list	0.668	0.5	0.276	3.137	1.86	8.223
+ fetching	21.517	7.948	2.829	0.054	1.052	1.769

Table 2.1: Component-wise breakdown of CPU time in seconds

The CPU time required for *transaction logging* is—even if the transaction throughput is not higher—much higher than in the case of single-threaded execution. This is caused by synchronization overhead in the log manager.

The increase in CPU time used in multithreaded execution by the *buffer pool manager* and in the *B-tree* is comparable to that of read-only *YCSB*.

2.3.4.3 TPC-C

Figure 2.18 shows the CPU time breakdown of *Zero* when running *TPC-C* with 24 working thread and a DB smaller than the buffer pool.

The overhead due to *transaction logging* is here—as it is only 8.45 % of the total CPU time—comparatively small, given the drastic growth in transaction throughput when ran on 24 working threads. The lower transaction throughput of *TPC-C* compared to *update-only YCSB* causes less synchronization overhead in the log manager, and with the higher transaction throughput of multithreaded compared to single-threaded *TPC-C* execution and with asynchronous commits, more log entries can be written at once.

The explosive growth in *B-tree* overhead compared to the single-threaded *TPC-C* execution can only partially be explained by the higher transaction throughput and overhead caused by the cache coherency protocol and the NUMA (Non-Uniform Memory Access) architecture, where pages are moved between memory banks of different CPUs and caches of different CPU cores. Other unknown effects must cause this overhead.

2.4 Optimizations Based on Profiling

The results from section 2.3 do not suggest a *single component* of *Zero* that should be optimized due to outstanding overhead at any workload. But Graefe et al. decided to optimize the buffer pool by saving calls to the hash table that uses 1.08 %–4.56 % of the CPU time. This optimization is evaluated in the subsection 2.4.1. A more detailed look at the analysis provided by the *VTune™ Profiler* suggests many *smaller optimizations*—a very simple one is—as an example—evaluated in subsection 2.4.2.

2.4.1 Pointer Swizzling

2.4.1.1 Definition

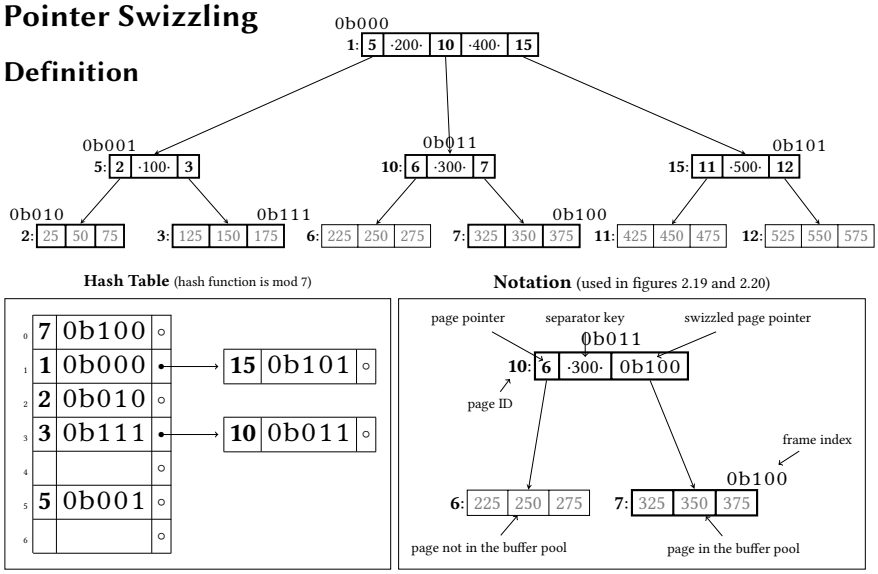


Figure 2.19: Example of a B+tree-like data structure partially in the buffer pool where the buffer pool manager uses a hash table as translation table.

To support *pointers to persistent objects* (such as DB pages), each persistent object requires an *identifier* (such as a page ID). A pointer can then simply be the ID of the corresponding persistent object. If the ID only needs to be unique within a file, the ID could simply be a *byte offset* in that file. So accessing an object based on such an ID would only require opening the corresponding file and reading the file from the specific byte offset. However, *logical IDs* that require data structures for the translation of these

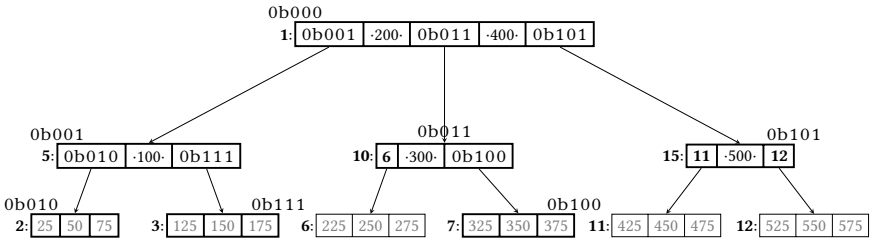


Figure 2.20: The same data structure and data like in figure 2.19: The buffer pool manager now uses pointer swizzling to locate pages.

IDs to a location in a file on secondary storage are more common in DBMSs. In figure 2.19, the root page of the B+tree-like data structure contains three such pointers to DB pages—**5**, **10** and **15**.

Due to the *access latency* of secondary storage devices, these *redirections via translation tables* do not represent a large *overhead* when accessing persistent objects. However, if the target of such a pointer is *already buffered in main memory*—like some of the DB pages in figure 2.19—to speed up access to it, finding the object based on these persistent IDs can become a potential bottleneck. This is especially true if almost all accessed objects are in main memory, i.e. the hit rate is very high. A *translation table* that maps persistent IDs to main memory addresses must be accessed—if the persistent ID of an object is contained in the table, it is buffered in main memory, otherwise it must be fetched from its original location in secondary memory. In figure 2.19 the translation table is implemented as *hash table* and maps the page IDs of buffered pages to corresponding memory addresses. For example, it maps page ID **7** to memory address (represented by a buffer index) `0b100`, since DB page **7** is buffered there.

Pointer swizzling is a technique used to speed up access to persistent objects buffered in main memory. Swizzling a pointer means replacing the identifier of the persistent object referenced there by a more direct address (usually the main memory address) of the transient object (the replica of the object in main memory) in such a way that this replacement can be used during several indirections of this pointer [Mos92].

Graefe et al. proposed in [Gra+14] the use of this technique for *Foster B-tree* pages buffered in a *DBMS buffer pool*. Figure 2.20 shows the same

B+tree-like data structure as figure 2.19, but with pointer swizzling used by the buffer pool. Since in the example all inner B+tree pages are in the buffer pool, all page IDs in the transient copy of the root page have been replaced by the buffer frame indices of the transient copies of the respective inner B+tree pages. Two inner pages also contain swizzled pointers.

Whenever a B+tree page is *going to be fixed*, this is done based on the pointer to that page contained in the corresponding parent page. *Without pointer swizzling*, this pointer is always the page ID, so the hash table must be probed to find out if the page is in the buffer pool—if not, it would be fetched into a free buffer frame and the mapping from the page ID to the buffer frame index would be added to the hash table.

With pointer swizzling, the buffer pool manager must simply check whether the pointer read from the transient copy of the parent page is a swizzled. A *flag bit* in the pointer is used to distinguish between buffer frame indices and page IDs—*if it is set*, the page is in the buffer pool, and the pointer (without the flag bit) represents the buffer frame index which can be used to access the page without further overhead.

If the *pointer is not swizzled*, the hash table still maintained here is probed to find out if the pointer is simply not swizzled although the page is in the buffer pool. If this is the case, the buffer manager swizzles the pointer in the parent page based on the buffer frame index from the hash table and fixes the page. If the page *is not contained in the buffer pool*, it is fetched into a free buffer frame, the corresponding mapping is added to the hash table, the pointer in the parent page is swizzled and the page is fixed.

To *write back a dirty B+tree page containing swizzled pointers*, these pointers must be unswizzled by the page cleaner, because swizzled pointers become invalid as soon as the corresponding page is evicted from the buffer pool. If a *page gets evicted*, the corresponding pointer in the parent page is unswizzled and the mapping is removed from the hash table.

2.4.1.2 Performance Evaluation

A performance evaluation of this pointer swizzling technique was already performed for my Bachelor's thesis [Gil17], but instabilities in the page eviction module of the buffer pool led to inaccurate measurement results.

Figure 2.21 shows new measurement results for a buffer pool using RAN-

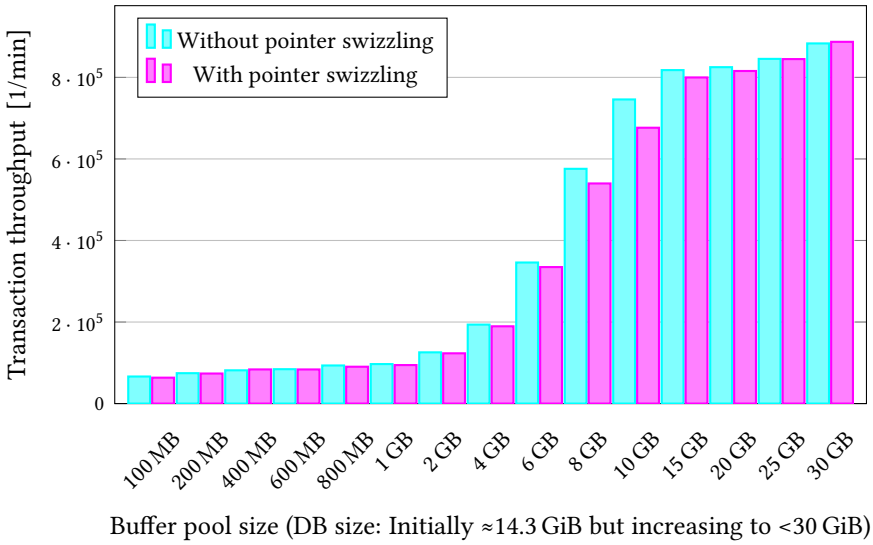


Figure 2.21: Transaction throughput of the RANDOM page replacement algorithm with and without pointer swizzling in the buffer pool for the *TPC-C* benchmark on 100 warehouses

DOM page replacement with and without pointer swizzling. Although the *hit rate* with pointer swizzling was consistently higher (which is unexpected) due to limitations added to the page replacement module by the implementation of pointer swizzling, the *transaction throughput* with pointer swizzling is not higher but actually lower for all buffer pool sizes. This is especially surprising for the 30 GB buffer pool, since there is no overhead caused by the swizzling and unswizzling of pointers, but only the saving of ≈4 % of CPU cycles that would otherwise be spent in the buffer pool. *Profiling* using the method used in section 2.3 also shows a significant reduction in CPU cycles per transaction, but does not provide an explanation for the poorer performance.

2.4.2 System Call: bzero

Finding *small blocks of code* that take up an *unexpectedly large amount of computing time* (or other resources) is probably the most common and basic

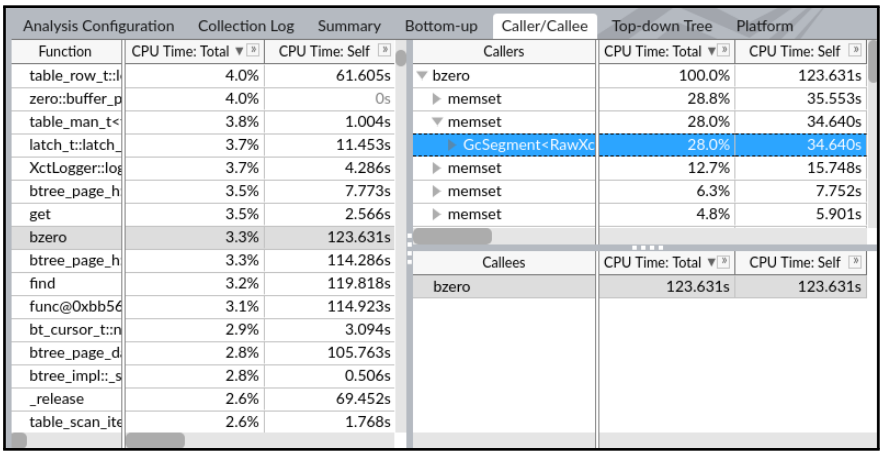


Figure 2.22: The "Caller/Callee" tab of an *Intel® VTune™ Profiler* analysis can be used to find code that consumes a significant amount of CPU time.

application of software tracing and profiling.

The screenshot in figure 2.22 shows part of the "Caller/Callee" tab of the *Intel® VTune™ Profiler* analysis of the multithreaded execution of *TPC-C* on *Zero*. The DBS spends 3.3 % (including benchmark code etc.) of CPU time in bzero, which *erases memory areas*. Such a call is a good candidate to find unnecessarily used CPU cycles, because such a call is not needed if e.g. the respective memory area is not reused without initialization.

```
1  struct GcSegment {
2      /* ... */
3      void recycle() {
4              owner = 0;
5              allocated_objects = 0;
6              ::memset(objects, 0, sizeof(T) * total_objects);
7      }
8      /* ... */
9  }
```

Listing 2.4: The member function GcSegment::recycle does unnecessarily erase a memory area.

This is the case for the caller GcSegment<RawXct>::recycle which is responsible for 28 % of the CPU time spent running bzero. Listing 2.4 shows the implementation of this member function. As soon as an object of class GcSegment is no longer needed, it is prepared for later reuse by overwriting all its member variables with zero. bzero is called by memset on line 7 because it is called to write zeros into a given memory area. But since it can be guaranteed that no part of the member variable objects is accessed after reuse before it is reinitialized, this call is unnecessary and a waste of CPU time.

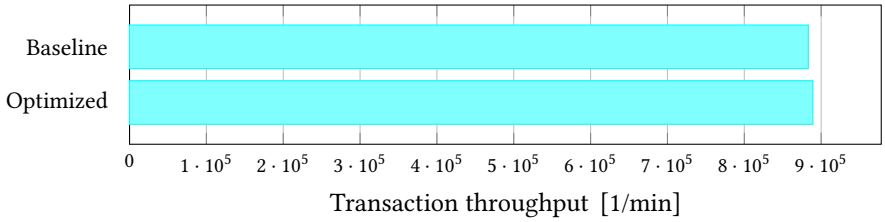


Figure 2.23: Transaction throughput of the RANDOM page replacement algorithm before and after removing unnecessary calls to bzero (memory erasure) for the *TPC-C* benchmark on 100 warehouses

Figure 2.22 shows the transaction throughput of *Zero* when running the *TPC-C* Benchmarks before and after removing the mentioned call of bzero. Both versions use the RANDOM page replacement algorithm and do not use the pointer swizzling technique from subsection 2.4.1. After saving these $3.3\,\% \cdot 28\,\% = 0.924\,\%$ of CPU cycles a $\approx 0.67\,\%$ higher transaction throughput can be achieved.

2.5 Conclusion

While the measurements from "OLTP through the Looking Glass, and What We Found There" were driven by the idea of *omitting certain guarantees and features* provided by relational DBMSs, the measurements for section 2.3 had the goal of *optimizing certain components* of a DBMS. Using the *tracing* and *profiling* tool *Intel® VTune™ Profiler* it was shown that, depending on the workload, different DBMS components require the largest proportion of CPU cycles. When many small transactions are executed, the transaction

manager becomes a bottleneck, but with a balanced workload such as *TPC-C*, key comparisons in the B-tree consume most CPU time.

These results suggest an optimization of the *transaction manager*. Since most of the CPU time was spent on spinning a spinlock, the complete removal of such a global latch would be an enormous optimization, making this component much more scalable. Optimizations in the *B-tree* are—as demonstrated here again—also always beneficial, especially when executing *TPC-C*. Even though *Zero* uses the well optimized Foster B-tree—many other approaches to optimize B+trees are known. Some of them are described by Goetz Graefe in his book "Modern B-tree techniques" [Gra04].

When executing *TPC-C*, the *buffer pool* used unexpectedly much CPU time, considering that (almost) all accessed DB pages were in memory during the whole measurement. Therefore, an optimization of the buffer pool like the *pointer swizzling* technique proposed by Graefe et al. in [Gra+14] looks promising. However, the performance evaluation of the technique in subsection 2.4.1 showed a decrease in transaction throughput when applied. However, subsection 2.4.2 showed that *removing a single line of code* can be enough to improve transaction throughput by ≈0.67 %, which is remarkable given the ease of the change. But as long as the quality of the implementation of a DBMS is not extremely poor, not much unnecessary code should be found that requires a considerable amount of CPU cycles, so such success should not be easily repeatable.

Bibliography

[AFJ00] Martin Arlitt, Rich Friedrich, and Tai Jin. "Performance Evaluation of Web Proxy Cache Replacement Policies". In: *Performance Evaluation* 39.1–4 (Feb. 2000). Ed. by Werner Bux and Ramon Puijaner, pp. 149–164. ISSN: 0166-5316. DOI: 10.1016/S0166-5316(99)00062-0.

[Arl+00] Martin Arlitt et al. "Evaluating Content Management Techniques for Web Proxy Caches". In: *SIGMETRICS Performance Evaluation Review* 27.4 (Mar. 2000). Ed. by Scott T. Leutenegger, pp. 3–11. ISSN: 0163-5999. DOI: 10.1145/346000.346003. URL: https://www.hpl.hp.com/techreports/98/HPL-98-173.pdf (visited on Aug. 15, 2020).

[BM04] Sorav Bansal and Dharmendra S. Modha. "CAR: Clock with Adaptive Replacement". In: *Proceedings of the 3rd USENIX Conference on File and Storage Technologies* (Mar. 31–Apr. 2, 2004). FAST '04. San Francisco, CA, USA: USENIX Association, Mar. 2004, pp. 187–200. URL: https://www.usenix.org/legacy/publications/library/proceedings/fast04/tech/full_papers/bansal/bansal.pdf (visited on Aug. 1, 2020).

[BM70] R. Bayer and E. McCreight. "Organization and Maintenance of Large Ordered Indices". In: *Proceedings of the 1970 ACM SIGFIDET Workshop on Data Description, Access and Control* (Nov. 15–16, 1970). SIGFIDET '70. Houston, TX, USA: Association for Computing Machinery, Nov. 1970, pp. 107–141. ISBN: 978-1-4503-7941-0. DOI: 10.1145/1734663.1734671.

[Bél66] László A. Bélády. "A Study of Replacement Algorithms for Virtual-Storage Computer". In: *IBM Systems Journal* 5 (2 June 1966), pp. 78–101. ISSN: 0018-8670. DOI: 10.1147/sj.52.0078.

[BNK16] Andrew Butterfield, Gerard Ekembe Ngondi, and Anne Kerr. *A Dictionary of Computer Science.* 7th. USA: Oxford University Press, Inc., 2016. ISBN: 978-0-19-968897-5. DOI: 10.1093/acref/9780199688975.001.0001.

[Cor69] F. J. Corbató. "A Paging Experiment with the Multics System". In: *In Honor of Philip M. Morse.* Ed. by Herman Feshbach and K. Uno Ingard. 1st ed. MIT Press Classic. Cambridge, MA, USA and London, UK: The M.I.T. Press, Nov. 15, 1969, pp. 217–228. ISBN: 978-0-262-06028-8. URL: https://www.multicians.org/paging-experiment.pdf (visited on Aug. 3, 2020).

[EH84] Wolfgang Effelsberg and Theo Härder. "Principles of Database Buffer Management". In: *ACM Transactions on Database Systems* 9 (4 Dec. 1984). Ed. by Robert W. Taylor, pp. 560–595. ISSN: 0362-5915. DOI: 10.1145/1994.2022.

[Gil17] Max Gilbert. "Evaluation of Pointer Swizzling Techniques for DBMS Buffer Management". BA thesis. Technische Universität Kaiserslautern, Mar. 1, 2017. URL: http://wwwlgis.informatik.uni-kl.de/cms/fileadmin/publications/2017/BScThesis_MaxGilbert.pdf (visited on July 20, 2020).

[Gil20] Max Gilbert. *Performance Evaluation of Different Open- Source Implementations of Data Structures and Algorithms in the Context of a DBMS Buffer Manager.* Research rep. Technische Universität Kaiserslautern, Jan. 15, 2020. URL: http://wwwlgis.informatik.uni-kl.de/cms/fileadmin/publications/2020/thesis.pdf (visited on July 3, 2020).

[Gra07] Goetz Graefe. "The Five-Minute Rule Twenty Years Later, and How Flash Memory Changes the Rules". In: *Proceedings of the 3rd International Workshop on Data Management on New Hardware* (June 15, 2007). DaMoN '07. Beijing, China: Association for Computing Machinery, June 2007, pp. 1–9. ISBN: 978-1-59593-772-8. DOI: 10.1145/1363189.1363198.

[Gra04] Goetz Graefe. *Modern B-Tree Techniques.* Vol. 3. Foundations and Trends in Databases 4. Hanover, MA, USA: Now Publishers Inc., 2011-04, pp. 203–402. DOI: 10.1561/1900000028.

[GKK12] Goetz Graefe, Hideaki Kimura, and Harumi Kuno. "Foster B-Trees". In: *ACM Transactions on Database Systems* 37 (3 Aug. 2012). Ed. by Zehra Meral Özsoyoğlu, 17:1–17:29. ISSN: 0362-5915. DOI: 10.1145/2338626.2338630.

[Gra+14] Goetz Graefe et al. "In-Memory Performance for Big Data". In: *Proceedings of the 41st International Conference on Very Large Data Bases.* Vol. 8: *Proceedings of the VLDB Endowment* (Aug. 31–Sept. 4, 2015). Ed. by Chen Li and Volker Markl. founding H. V. Jagadish. In collab. with Kevin Chang et al. Proceedings of the VLDB Endowment 1. Very Large Data Base Endowment Inc. Kohala Coast, HI, USA, Sept. 2014, pp. 37–48. ISSN: 2150-8097. DOI: 10.14778/2735461.2735465. URL: http://www.vldb.org/pvldb/vol8/p37-graefe.pdf (visited on Dec. 13, 2016).

[GG97] Jim Gray and Goetz Graefe. "The Five-Minute Rule Ten Years Later, and Other Computer Storage Rules of Thumb". In: *SIGMOD Record* 26.4 (Dec. 1997). Ed. by Michael Franklin, pp. 63–68. ISSN: 0163-5808. DOI: 10.1145/271074.271094.

[GP87] Jim Gray and Franco Putzolu. "The 5 Minute Rule for Trading Memory for Disc Accesses and the 10 Byte Rule for Trading Memory for CPU Time". In: *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data.* SIGMOD '87. San Francisco, CA, USA: Association for Computing Machinery, Dec. 1987, pp. 395–398. ISBN: 978-0-89791-236-5. DOI: 10.1145/38713.38755.

[Har+08] Stavros Harizopoulos et al. "OLTP through the Looking Glass, and What We Found There". In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (June 10–12, 2008). SIGMOD '08. Vancouver, Canada: Association for Computing Machinery, June 2008, pp. 981–992. ISBN: 978-1-60558-102-6. DOI: 10.1145/1376616.1376713. URL: http://nms.csail.mit.edu/~stavros/pubs/OLTP_sigmod08.pdf (visited on Aug. 20, 2020).

[JCZ05] Song Jiang, Feng Chen, and Xiaodong Zhang. "CLOCK-Pro: An Effective Improvement of the CLOCK Replacement". In:

Proceedings of the Annual Conference on USENIX Annual Technical Conference (Apr. 10–15, 2005). ATEC '05. Anaheim, CA, USA: USENIX Association, Apr. 2005, pp. 323–336. URL: https://rcs.uwaterloo.ca/papers/clockpro.pdf (visited on Aug. 1, 2020).

[JZ02] Song Jiang and Xiaodong Zhang. "LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance". In: *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems* (June 15–19, 2002). SIGMETRICS '02. Marina Del Rey, CA, USA: Association for Computing Machinery, June 2002, pp. 31–42. ISBN: 978-1-58113-531-2. DOI: 10.1145/511334.511340.

[Joh+09] Ryan Johnson et al. "Shore-MT: A Scalable Storage Manager for the Multicore Era". In: *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology* (Mar. 23–26, 2009). Ed. by Martin Kersten et al. EDBT '09. Saint Petersburg, RU: Association for Computing Machinery, Mar. 2009, pp. 24–35. ISBN: 978-1-60558-422-5. DOI: 10.1145/1516360.1516365.

[JS94] Theodore Johnson and Dennis Shasha. "2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm". In: *Proceedings of the 20th International Conference on Very Large Data Bases* (Sept. 12–15, 1994). VLDB '94. Santiago de Chile, Chile: Morgan Kaufmann Publishers Inc., Sept. 1994, pp. 439–450. ISBN: 978-1-55860-153-6.

[KLW94] Ramakrishna Karedla, J. Spencer Love, and Bradley G. Wherry. "Caching Strategies to Improve Disk System Performance". In: *Computer* 27.3 (Mar. 1994), pp. 38–46. ISSN: 0018-9162. DOI: 10.1109/2.268884.

[LKN13] Viktor Leis, Alfons Kemper, and Thomas Neumann. "The Adaptive Radix Tree: ARTful Indexing for Main-Memory Databases". In: *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE 2013)* (Apr. 8–11, 2013). ICDE '13. Brisbane, AU: IEEE Computer Society, Apr. 2013, pp. 38–49.

ISBN: 978-1-4673-4909-3. DOI: 10.1109/ICDE.2013.6544812. URL: https://db.in.tum.de/~leis/papers/ART.pdf (visited on Aug. 22, 2020).

[Lei+18] Viktor Leis et al. "LeanStore: In-Memory Data Management beyond Main Memory". In: *34th IEEE International Conference on Data Engineering* (Apr. 16–19, 2018). ICDE '18. Paris, France: IEEE Computer Society, Apr. 2018, pp. 185–196. ISBN: 978-1-5386-5520-7. ISSN: 2375-026X. DOI: 10.1109/ICDE.2018.00026. URL: https://db.in.tum.de/~leis/papers/leanstore.pdf (visited on Sept. 15, 2020).

[Li18] Cong Li. "DLIRS: Improving Low Inter-Reference Recency Set Cache Replacement Policy with Dynamics". In: *Proceedings of the 11th ACM International Systems and Storage Conference* (June 4–6, 2018). SYSTOR '18. Haifa, Israel: Association for Computing Machinery, June 2018, pp. 59–64. ISBN: 978-1-4503-5849-1. DOI: 10.1145/3211890.3211891. URL: https://www.systor.org/2018/pdf/systor18-4.pdf (visited on Aug. 1, 2020).

[Li19] Cong Li. "CLOCK-Pro+: Improving CLOCK-pro Cache Replacement with Utility-Driven Adaptation". In: *Proceedings of the 12th ACM International Conference on Systems and Storage* (June 3–5, 2019). SYSTOR '19. Haifa, Israel: Association for Computing Machinery, June 2019, pp. 1–7. ISBN: 978-1-4503-6749-3. DOI: 10.1145/3319647.3325838.

[MM03] Nimrod Megiddo and Dharmendra S. Modha. "ARC: A Self-Tuning, Low Overhead Replacement Cache". In: *Proceedings of the 2nd USENIX Conference on File and Storage Technologies* (Mar. 31–Apr. 2, 2003). FAST '03. San Francisco, CA, USA: USENIX Association, Mar. 2003, pp. 115–130. URL: https://www.usenix.org/legacy/events/fast03/tech/full_papers/megiddo/megiddo.pdf (visited on Aug. 1, 2020).

[Mos92] J. Eliot B. Moss. "Working with Persistent Objects: To Swizzle or Not to Swizzle". In: *IEEE Transactions on Software Engineering* 18.8 (Aug. 1992). Ed. by Nancy G. Leveson, pp. 657–673. ISSN: 0098-5589. DOI: 10.1109/32.153378.

[OOW93] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. "The LRU-K Page Replacement Algorithm for Database Disk Buffering". In: *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data* (May 1993). Ed. by Peter Buneman and Sushil Jajodia. SIGMOD. Washington, D.C., USA: Association for Computing Machinery, 1993, pp. 297–306. ISBN: 978-0-89791-592-2. DOI: 10.1145/170035.170081. URL: https://www.cs.cmu.edu/~christos/courses/721-resources/p297-o_neil.pdf (visited on July 23, 2020).

[PA16] Andrew Pavlo and Matthew Aslett. "What's Really New with NewSQL?" In: *SIGMOD Record* 45.2 (June 2016). Ed. by Yanlei Diao, pp. 45–55. ISSN: 0163-5808. DOI: 10.1145/3003665.3003674. URL: https://db.cs.cmu.edu/papers/2016/pavlo-newsql-sigmodrec2016.pdf (visited on Aug. 26, 2020).

[Pri08] Dan Pritchett. "BASE: An Acid Alternative". In: *Queue* 6.3 (May 2008), pp. 48–55. ISSN: 1542-7730. DOI: 10.1145/1394127.1394128.

[Ree78] David P. Reed. "Naming and Synchronization in a Decentralized Computer System". PhD thesis. Massachusetts Institute of Technology, Sept. 21, 1978. URL: http://www.lcs.mit.edu/publications/pubs/pdf/MIT-LCS-TR-205.pdf (visited on Sept. 10, 2020).

[Smi78] Alan Jay Smith. "Sequentiality and Prefetching in Database Systems". In: *ACM Transactions on Database Systems* 3.3 (Sept. 1978). Ed. by David K. Hsiao. In collab. with Rudolf Bayer et al., pp. 223–247. ISSN: 0362-5915. DOI: 10.1145/320263.320276.

[Töz+13] Pınar Tözün et al. "From A to E: Analyzing TPC's OLTP Benchmarks: The Obsolete, the Ubiquitous, the Unexplored". In: *Proceedings of the 16th International Conference on Extending Database Technology* (Mar. 18–22, 2013). Ed. by Norman W. Paton et al. EDBT '13. Genoa, Italy: Association for Computing Machinery, Mar. 2013, pp. 17–28. ISBN: 978-1-4503-1597-5. DOI: 10.1145/2452376.2452380. URL: https://infoscience.epfl.ch/record/182894/files/ptozun_edbt2013.pdf (visited on Sept. 9, 2020).

[Wu+17] Yingjun Wu et al. "An Empirical Evaluation of In-Memory Multi-Version Concurrency Control". In: *Proceedings of the 43rd International Conference on Very Large Data Bases.* Vol. 10: *Proceedings of the VLDB Endowment* (Aug. 28–Sept. 1, 2017). Ed. by Peter Boncz and Ken Salem. founding H. V. Jagadish. Proceedings of the VLDB Endowment 7. Very Large Data Base Endowment Inc. Munich, BY, DE, Mar. 2017, pp. 781–792. ISSN: 2150-8097. DOI: 10.14778/3067421.3067427. URL: http://www.vldb.org/pvldb/vol10/p781-Wu.pdf (visited on Sept. 10, 2020).

[ZPL01] Yuanyuan Zhou, James Philbin, and Kai Li. "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches". In: *Proceedings of the General Track: 2001 USENIX Annual Technical Conference* (June 25–30, 2001). Boston, MA, USA: USENIX Association, June 2001, pp. 91–104. ISBN: 978-1-880446-09-6. URL: https://static.usenix.org/event/usenix01/full_papers/zhou/zhou.pdf (visited on Aug. 15, 2020).