TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Department of Computer Science
Database and Information Systems Group

Project Thesis:

Performance Evaluation of Different Open Source Implementations of Data Structures and Other Algorithms in the context of a DBMS Buffer Manager

by **Max Fabian Gilbert**[*]

Day of release: January 31, 2019

[*]m_gilbert13@cs.uni-kl.de

Abstract

Needless to say, every database management system needs to be able to manage data. The data structures used to manage those data in a database have a major influence on various characteristics (e.g. performance) of a database management system and therefore, the usage of specific data structures (e.g. B-tree indexes) and even some implementation details of those are very important decisions in DBMS design.

But for correct and performant operation, a DBMS needs to manage various kinds of meta data as well. Some of those meta data needs to be persistent (e.g. the catalog of a relational DBMS) but some can also be non-persistent. Because of the non-persistence of data managed by the buffer management of a DB, the meta data required for the buffer manager are also usually non-persistent. The data structures used to manage those meta data are—unlike the data structures used to manage the data—more an implementation than a design decision. For some kinds of those meta data, it's—due to the non-criticality of the specific meta data management—even reasonable to use data structures provided by the used programming language even though there might be more performant data structures for the purpose. But more performant implementations for most of those data structures don't need to be implemented specifically for one project, there are many different implementations available in open source and proprietary libraries.

This work is a performance evaluation of various MPMC

Contents

1 Buffer Frame Free List

1.1 Purpose

When not relying on the unsuitable VM management of the OS—every disk-based DBMS requires some kind of buffer manager which provides in-memory copies of database pages—which are stored persistently on secondary storage—to the upper layers of the DBMS for processing.

This feature is provided by the buffer pool management by managing the currently used subset of the database pages—the working set—in buffer frames located in memory. In the common case of fixed-size pages, a buffer frame is a portion of memory that can hold one database page and each of those frames has a frame index as identifier.

During operation, database pages are dynamically fetched from the database into buffer frames. Once a page's availability in memory is not required anymore (i.e. it is not required for the processing of any current transaction), it might be evicted from the buffer pool freeing a buffer frame.

The buffer manager needs some kind of free list when it allocates a buffer frame to a fetched database page because unrestricted overwriting of data in buffer frames would cause undefined behavior.

1.2 Compared Open Source Queue Implementations

To ease implementation of page eviction strategies like CLOCK, a free list should use a FIFO data structure like a queue—typically implemented as linked list. Therefore, the buffer frame freed first is (re-)used first as well.

Almost every state-of-the-art DBMS supports multi-threading and therefore, there are usually multiple threads concurrently fetching pages into the buffer pool and evicting pages from the buffer pool. Following this, a buffer frame free list has to support thread-safe functions to push frame indexes to the free list and to pop frame indexes from it. Queues providing those thread-safe access functions are usually called multi-producer (add frame indexes) multi-consumer (retrieve/remove frame indexes) queues (**MPMC** queues).

An approximate number of buffer indexes in the free list should also be provided by any free list implementation to support the (batch-wise) eviction of pages once there are only a few free buffer frames left. Thread-safe access to this number is desirable but not absolutely required.

1.2.1 Boost Lock-Free Queue with Variable Size

The popular *Boost C++ Libraries*[1] offer a **lock-free unbounded** MPMC queue[2] in the library `Boost.Lockfree`[3].

Like many other **non-blocking** thread-safe data structures, this MPMC queue uses atomic operations instead of locks or mutexes. To support dynamic growing and shrinking of the queue, this queue implementation also uses a free list for its own internal dynamic memory management.

This data structure does not offer the number of contained elements and therefore, an approximate number of buffer indexes in the free list needs to be managed externally.

1.2.2 Boost Lock-Free Queue with Fixed Size

This queue implementation is identical to the data structure in Subsection 1.2.1 but does not use dynamic memory management internally—it is a **bounded** queue. As long as the needed capacity of the queue—in our case the maximum number of buffer frames in the buffer pool—is known when the queue is allocated, this queue implementation can be used. This implementation uses a fixed-size array instead of dynamically allocated nodes for its stored data—the indexes of free buffer frames.

[1]`https://www.boost.org/`
[2]`https://bit.ly/2Q9w45H`
[3]`https://bit.ly/2FiPyip`

1.2.3 CDS Basket Lock-Free Queue

Besides many other concurrent data structures, the *Concurrent Data Structures* C++ library[4] offers many different thread-safe queue implementations. The **unbounded** *basket lock-free queue*[5] is based on the algorithm proposed by M. Hoffman, O. Shalev and N. Shavit in [HSS07].

Internally, this queue does not use an absolute FIFO order. Instead, it puts concurrently enqueued elements into one "basket" of elements. The elements within one basket are not specifically ordered but the different "baskets" used over time are ordered according to FIFO. Therefore, the dequeue operation just dequeues one of the elements in the oldest "basket". The dynamic memory management uses a garbage collector to deallocate emptied "baskets".

1.2.4 CDS Flat-Combining Sequential Queue

The *Concurrent Data Structures* C++ library does also offer an **unbounded** thread-safe queue that uses flat combining[6]. The flat combining technique was proposed by D. Hendler, I. Incze, N. Shavit and M. Tzafrir in [Hen+10]. This technique can make any sequential data structure thread-safe—in case of the *flat-combining sequential queue*, the `std::queue`[7] of the *C++ Standard Library*[8] is used as base data structure.

The flat combining technique uses thread-local publication lists to record operations performed by those threads. A **global lock** is needed to be acquired to combine these thread-local publication lists into the global, sequential data structure. The thread which acquired the global lock also combines the publication lists of all other threads reducing the locking overhead. The returned value of each operation executed during the combining is stored into the respective publication list together with the global combining pass number. A thread with a non-empty publication list that cannot acquire the global lock needs to wait till the combining thread updated its publication list.

[4]`https://github.com/khizmax/libcds`
[5]`https://bit.ly/2SGv2A6`
[6]`https://bit.ly/2ZEsrYO`
[7]`https://en.cppreference.com/w/cpp/container/queue`
[8]`https://en.cppreference.com/w/cpp`

1.2.5 CDS Michael & Scott Lock-Free Queue

Another **unbounded lock-free** queue implementation offered by the *Concurrent Data Structures* C++ library is based on the famous Michael & Scott lock-free queue algorithm[9], proposed by M. Michael and M. Scott in [MS96].

The Michael & Scott lock-free queue basically uses compare-and-swap (**CAS**) operations on the tail of the queue to synchronize enqueue operations. If a thread reads a NULL value as next element after the queue's tail, it swaps this value atomically with the value enqueued by this thread. Afterwards it adjusts the tail pointer. If a thread does not read the NULL value there during the CAS operation, another thread has not already adjusted the tail pointer and this thread needs to retry its enqueue operation with the new tail pointer. The dequeue operation is implemented similarly. The memory occupied by already dequeued elements is deallocated using a garbage collector provided by the library.

1.2.6 CDS Variation of Michael & Scott Lock-Free Queue

The *Concurrent Data Structures* C++ library also offers an optimized variation of the Michael & Scott **unbounded lock-free** queue algorithm[10] which is based on the works of S. Doherty, L. Groves, V. Luchangco and M. Moir in [Doh+04].

This optimization of the Michael & Scott lock-free queue optimizes the dequeue operation to only read the tail pointer once.

1.2.7 CDS Michael & Scott Blocking Queue with Fine-Grained Locking

M. Michael and M. Scott did also propose a blocking queue algorithm in [MS96]. This **unbounded blocking** queue implementation[11] is also offered by the *Concurrent Data Structures* C++ library.

This blocking queue algorithm uses one read and one write lock protecting the head and tail of the queue. Therefore, only one thread a time can enqueue and only one thread at a time can dequeue elements. The

[9] `https://bit.ly/37onMwC`
[10] `https://bit.ly/2MG8dbM`
[11] `https://bit.ly/2SCeFo5`

deallocation of memory during dequeuing is done by the dequeuing thread instead of relying on a garbage collector.

1.2.8 CDS Ladan-Mozes & Shavit Optimistic Queue

The *Concurrent Data Structures* C++ library also offers an **unbounded optimistic** queue implementation[12] which is based on an algorithm proposed by E. Ladan-Mozes and N. Shavit in [LS04].

Instead of using expensive CAS operations on a singly-linked list (like in the Michael & Scott lock-free queue), this algorithm uses a doubly-linked list with the possibility to detect and fix inconsistent enqueue and dequeue operations. Deallocation of memory is done using a garbage collector.

1.2.9 CDS Segmented Queue

The **unbounded** segmented queue implementation[13] of the *Concurrent Data Structures* C++ library is based on an algorithm proposed by Y. Afek, G. Korland and E. Yanovsky in [AKY10].

This thread-safe queue algorithm is very similar to the basket lock-free queue from Subsection 1.2.3. It also uses a relaxed FIFO order by ordering segments containing multiple elements instead of single elements. A thread enqueuing or dequeuing elements into the tail segment or from the head segment selects one of the slots inside the segment randomly. CAS operations are used to atomically enqueue or dequeue an element from a slot. If the CAS fails, another slot is taken randomly. The size of each segment—which can be selected (8 was used for the performance evaluation in Section 1.3)—determines the relaxness of the FIFO order. Deallocation of emptied segments is done by a garbage collector.

1.2.10 CDS Vyukov's MPMC Bounded Queue

The last thread-safe queue implementation[14] provided by the *Concurrent Data Structures* C++ library is **bounded** and was developed by D. Vyukov[15].

[12]`https://bit.ly/37mgQQM`
[13]`https://bit.ly/37mjXYR`
[14]`https://bit.ly/2ML9Y7M`
[15]`https://bit.ly/39n4PMF`

The queue in Subsection 1.2.12 is his original implementation.

1.2.11 Folly MPMC Queue

Facebook's open source library *Folly*[16] provides a **bounded lock-free** queue implementation. An unbounded version is also provided but due to the typically higher performance of bounded ones, it is not evaluated here.

Folly's MPMC queue uses a ticket dispenser system to give a thread access to one of the single-element queues used. Those ticket dispensers for the head and tail of the queue use atomic increment operations which are supposed to be more robust to contention than CAS operations used e.g. in the Michael & Scott lock-free queue.

1.2.12 Dmitry Vyukov's Bounded MPMC Queue

This[17] is Vyukov's original implementation of his **bounded** thread-safe MPMC queue.

Vyukov's thread-safe queue implementation is very similar to Michael & Scott blocking queue with fine-grained locking from Subsection 1.2.7 but instead of using mutexes as locks, his implementation uses atomic read-modify-write (**RMW**) operations. This results in a cost of basically one CAS operation per enqueue/dequeue operation.

1.2.13 Gavin Lambert's MPMC Bounded Lock-Free Queue

This[18] is another version of Vyukov's thread-safe queue design from Subsection 1.2.12 implemented by Gavin Lambert.

1.2.14 Matt Stump's Bounded MPMC Queue

This[19] is another version of Vyukov's thread-safe queue design from Subsection 1.2.12 implemented by Matt Stump.

[16]`https://github.com/facebook/folly`
[17]`https://bit.ly/35a5lKL`
[18]`https://bit.ly/2F7jvlb`
[19]`https://github.com/mstump/queues`

1.2.15 Erik Rigtorp's Bounded MPMC Queue

The **bounded lock-free** queue[20] of Erik Rigtorp uses a ticket dispenser system similar to the one of *Folly*'s MPMC queue from Subsection 1.2.11.

1.2.16 TBB Concurrent Queue

The *Threading Building Blocks* library[21] is an open source library originally developed by Intel®. The first thread-safe queue implementation[22] of this library is **unbounded** and **non-blocking**.

Internally, this queue implementation uses multiple lock-based micro queues to allow concurrent enqueue/dequeue executions. Therefore, this thread-safe queue does not maintain the order of elements enqueued by different threads.

Due to the implementation using multiple SPMC queues, this queue implementation should only be used as a buffer frame free list when there is exactly one thread evicting pages from the buffer pool—and therefore, enqueuing buffer frame indexes of emptied buffer frames.

1.2.17 TBB Bounded Concurrent Dual Queue

The other thread-safe queue implementation[23] of the *Threading Building Blocks* library is **bounded** and **partially non-blocking**.

This queue implementation is almost identical to the other one of the *Threading Building Blocks* library but it does allow the limitation of the capacity. An enqueuing operation has to wait if the queue is already full according to the specified capacity.

[20] `https://github.com/rigtorp/MPMCQueue`
[21] `https://www.threadingbuildingblocks.org/`
[22] `https://software.intel.com/en-us/node/506200`
[23] `https://software.intel.com/en-us/node/506201`

1.3 Performance Evaluation

1.3.1 Microbenchmark

The used microbenchmark simulates a high contented free list. The number of working threads, the number of iterations (either the fetching of a page into a free buffer frame or the eviction of a batch of pages) per thread and the batch size of buffer frames to be freed at once can be varied. It does not simulate a complete buffer pool—there is only the free list with operations to enqueue and dequeue buffer frame indexes. Each working thread performs the following operations per iterations:

- If the free list is not empty:
  - Retrieve a buffer frame index from the free list.
  - Mark the retrieved buffer frame used.
- If the free list is empty:
  - While the free list is smaller than the batch eviction size:
    - Select a random buffer frame index using a fast random numbers generator.
    - If this buffer frame index is marked used:
      - Mark the selected buffer frame index unused.
      - Add the selected buffer frame index to the free list.

1.3.2 Used Versions of the Libraries and Queue Implementations

- *Boost C++ Libraries* 1.67
- *Concurrent Data Structures* C++ library 2.3.1[24]
- *Folly* `a8d1fd8`[25]
- Dmitry Vyukov's Bounded MPMC Queue as of September 2017[26]

[24]`https://bit.ly/39vysvB`
[25]`https://bit.ly/2sy4J4l`
[26]`https://bit.ly/2MHbXtG`

- Gavin Lambert's MPMC Bounded Lock-Free Queue `e409068`[27]
- Matt Stump's MPMC Queue `319c253`[28]
- Erik Rigtorp's MPMC Queue `553cf42`[29]
- Intel® Threading Building Blocks 2019 Update 9

1.3.3 Configuration of the Used System

- **CPU:** *Intel® Core™ i7-8700* @12 × 3.2 GHz from late 2017
- **Main Memory:** 2 × 8GB = 16GB of DDR4-SDRAM @2666 MHz
- **OS:** *Ubuntu 19.10*

1.3.4 Microbenchmark Results

Figure 1.1 shows the free list operation throughput measured with the microbenchmark. Each iteration of the microbenchmark from Subsection 1.3.1—performing either a pull or a push operation on the free list—is one operation on the evaluated free list queue. The operation throughput is total number of operations (from all working threads) performed on the free list per time.

The *CDS segmented queue* —●— is the slowest free list queue implementation for any number of concurrently working threads. The very similar *CDS basket lock-free queue* —▲— performs as bad on 1 working thread and not much better on a higher number of threads.

The *CDS* queue implementations *CDS Ladan-Mozes & Shavit optimistic queue* —▲—, *CDS Michael & Scott lock-free queue* —■— and *CDS variation of Michael & Scott lock-free queue* —●— are also similar to each other and therefore, they all perform very similarly. Their performance for a low number of working threads is better than the one of the *CDS Basket Lock-Free Queue* but for higher numbers of threads they perform even worse.

The very simple *CDS flat-combining lock-free queue* —◆— performs not too bad for ≤ 2 threads but the global lock limits the concurrency and

[27]`https://bit.ly/2sy6QoN`
[28]`https://bit.ly/2ZBEMgk`
[29]`https://bit.ly/2F7cH7d`

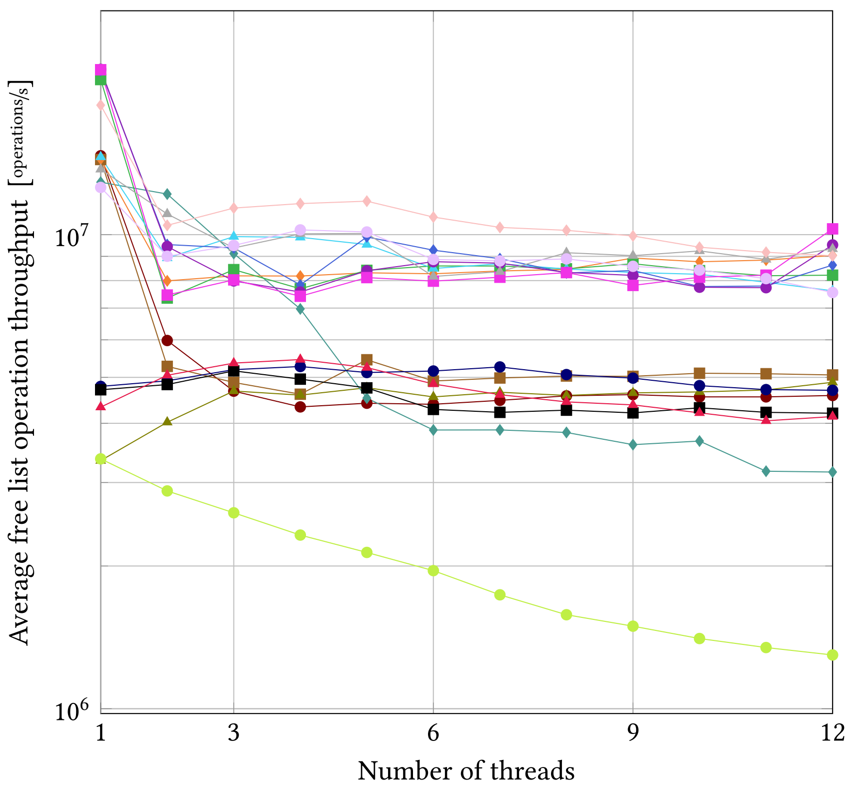

Figure 1.1: The operation throughput of the evaluated free list queue implementations

therefore, the performance falls by a factor of > 3 when there are ≥ 6 threads concurrently working on the free list queue.

The throughput of the two queue implementations from the *Boost C++ Libraries*—the *Boost lock-free queue with variable size* —●— and the *Boost lock-free queue with fixed size* —■—drops even earlier than the one of the *CDS flat-combining lock-free queue*. For one working thread, they perform good but for > 1 threads, their performance is bad. The overhead due to the dynamic memory management of the unbounded version is negligible.

The three implementations of Vyukov's MPMC queue design—*CDS Vyukov's MPMC bounded queue* —■—, *Matt Stump's bounded MPMC queue*

■ and *Gavin Lambert's MPMC bounded lock-free queue* ●—perform better than the ones mentioned already but the original *Dmitry Vyukov's bounded MPMC queue* ◆ performs even better. All the implementations of Vyukov's MPMC queue design are the best free list queues when there is only one working thread.

The two queue implementations which use a ticket dispenser system for synchronization—the *Folly MPMC queue* ▲ and *Erik Rigtorp's bounded MPMC queue* ▲—perform as good as Vyukov's MPMC queue design.

The *CDS Michael & Scott blocking queue with fine-grained locking* ◆ and *TBB bounded concurrent dual queue* ● perform very similar as well.

The best free list queue for > 2 threads concurrently operating on the free list is the *TBB concurrent queue* ◆.

1.4 Conclusion

The performance of the buffer pool free list is typically not critical for the overall performance of a DBMS—even a bad free list queue is unlikely to become the bottleneck of a DBMS. The buffer pool free list is mostly called when a database page is retrieved from secondary storage which is—even on an enterprise SSD—a very expensive operation.

Depending on the further developments in NVRAM technology, memory devices of this class might not be able to completely replace DRAM in database applications in the near future, but they might replace—and already do so—existing secondary storage technologies like SSDs. According to specifications from [APD15], the limiting factor—at least in OLTP applications—will be the endurance of the memory cells—which is basically the number of write operations per memory cell until it is worn out.

Following these general assumptions, high contention on the buffer pool free list is very unlikely and therefore, the usage of *Dmitry Vyukov's bounded MPMC queue* as buffer pool free list implementation can be recommended. But due to the superiority of the *TBB concurrent queue* during contention on the free list, the low drawback during non-concurrent operation and the maturity of the library, this queue implementation can also be recommended.

2 RANDOM Page Eviction

2.1 Purpose

The buffer manager of a DBMS needs to evict pages from buffer frames when currently not buffered pages need to be fetched from the database while there are no more free buffer frames. Every buffer manager got a page evictioner—implementing one of the many page eviction algorithms developed since the 1960s—for that purpose.

According to Belady's classification in [Bel66], the RANDOM eviction algorithm is the most representative algorithm in his *Class 1* of page eviction algorithms. Those *Class 1* page eviction algorithms do not use any information about the usage of a buffered page but just apply a static rule for the eviction decision. According to the newer classification of Effelsberg and Härder in [EH84], the RANDOM eviction algorithm is the only algorithm in the class of algorithms using neither the age of a buffered page nor the references of it for the eviction decision.

The RANDOM strategy is the simplest page eviction strategy possible resulting in a low overhead and bad hit rates.

2.2 Compared Pseudorandom Number Generators

The only operation performed by the RANDOM page evictioner to decide which page to eviction from the buffer pool is the generation of a pseudorandom number in the range of buffer frame indexes. The database page contained in the selected buffer frame is evicted afterwards.

There are many different classes of pseudorandom number generators (**PRNG**). Some of them provide pseudorandom numbers of high randomness others just take only few CPU cycles and almost no memory to generate a random number.

2.2.1 `std::minstd_rand0`

The *C++ Standard Library*[1] provides the original *MINSTD* PRNG as proposed by S. Park and K. Miller in [PM88] predefined in the header `<random>`[2].

The original *MINSTD* is a *Lehmer random number generator* which is a linear congruential generator (**LCG**). It uses the parameters $m = 2^31 - 1$, $a = 16807$ and $c = 0$.

2.2.2 `std::minstd_rand`

The *C++ Standard Library*[1] also provides the revised *MINSTD* PRNG as proposed by S. Park, K. Miller and P. Stockmeyer in [PMS93] predefined in the header `<random>`[2].

This revised *MINSTD* improves the quality of generated pseudorandom numbers by using the parameter $a = 48271$.

2.2.3 `std::mt19937`

The standard implementation of the Mersenne Twister MT19937—proposed by M. Matsumoto and T. Nishimura in [MN98]—is also provided in the header `<random>`[2] of the *C++ Standard Library*[1].

2.2.4 `std::mt19937_64`

The 64-bit version of the Mersenne Twister—MT19937-64—is also provided in the header `<random>`[2] of the *C++ Standard Library*[1].

2.2.5 `std::ranlux24_base`

This PRNG, based on the 24-bit RANLUX generator—proposed in [Lüs94] by M. Lüscher—, is a *Subtract-With-Borrow* generator provided by the *C++ Standard Library*[1] in header `<random>`[2].

[1] `https://en.cppreference.com/w/cpp`
[2] `https://en.cppreference.com/w/cpp/numeric/random`

The computational cost of this PRNG is very high but the pseudorandom numbers generated are of high randomness.

2.2.6 `std::ranlux48_base`

This PRNG is like the PNRG from Subsection 2.2.5 but it uses 48-bit instead of 24-bit words. It is provided by the *C++ Standard Library*[1] in header `<random>`[2].

2.2.7 `std::ranlux24`

This is the 24-bit RANLUX generator originally proposed by M. Lüscher and F. James in [Lüs94] and [Jam94] provided by the *C++ Standard Library*[1] in header `<random>`[2].

2.2.8 `std::ranlux48`

This is the 48-bit version of the PRNG from Subsection 2.2.7 provided by the *C++ Standard Library*[1] in header `<random>`[2].

2.2.9 `std::knuth_b`

2.2.10 `std::rand`

2.2.11 `std::random_device`

2.2.12 Xorshift32

```
thread_local bool seed_initialized;
thread_local uint32_t seed;

uint32_t xorshift32_random() {
    if (!seed_initialized) {
        seed = std::random_device{}();
        seed_initialized = true;
    }
    seed ^= seed << 13;
    seed ^= seed >> 17;
    seed ^= seed << 5;
    return (seed % (block_count - 1) + 1);
}
```

2.2.13 Xorshift64

```cpp
thread_local bool seed_initialized;
thread_local uint64_t seed;

uint32_t xorshift64_random() {
    if (!seed_initialized) {
        seed = std::random_device{}();
        seed_initialized = true;
    }
    seed ^= seed << 13;
    seed ^= seed >> 7;
    seed ^= seed << 17;
    return (seed % (block_count - 1) + 1);
}
```

2.2.14 Xorshift96

```cpp
thread_local bool seed_initialized;
thread_local uint32_t seed_0;
thread_local uint32_t seed_1;
thread_local uint32_t seed_2;

uint32_t xorshift96_random() {
    if (!seed_initialized) {
        seed_0 = std::random_device{}();
        seed_1 = std::random_device{}();
        seed_2 = std::random_device{}();
        seed_initialized = true;
    }
    uint32_t t;
    seed_0 ^= seed_0 << 16;
    seed_0 ^= seed_0 >> 5;
    seed_0 ^= seed_0 << 1;

    t = seed_0;
    seed_0 = seed_1;
    seed_1 = seed_2;
    seed_2 = t ^ seed_0 ^ seed_1;

    return (seed_2 % (block_count - 1)) + 1;
}
```

2.2.15 Xorshift128

```
thread_local bool seed_initialized;
thread_local uint32_t seed_0;
thread_local uint32_t seed_1;
thread_local uint32_t seed_2;
thread_local uint32_t seed_3;

uint32_t xorshift128_random() {
    if (!seed_initialized) {
        seed_0 = std::random_device{}();
        seed_1 = std::random_device{}();
        seed_2 = std::random_device{}();
        seed_3 = std::random_device{}();
        seed_initialized = true;
    }
    uint32_t t = seed_0 ^ (seed_0 << 11);
    seed_0 = seed_1;
    seed_1 = seed_2;
    seed_2 = seed_3;

    seed_3 ^= (seed_3 >> 19) ^ t ^ (t >> 8);

    return (seed_3 % (block_count - 1) + 1);
}
```

2.2.16 XorWow

2.2.17 Xorshift*

2.2.18 Xorshift+

2.2.19 Xoroshiro128+

2.3 Performance Evaluation

2.3.1 Micro Benchmark

2.3.2 Configuration of the Used System

- **CPU:** 2× *Intel® Xeon® Processor X5670* @6 × 2.93GHz released early 2010
- **Main Memory:** 12 × 8GB = 96GB of DDR2-SDRAM @1333MHz
- **OS:** *Ubuntu 16.04*

2.4 Conclusion

3 LOOP Page Eviction

3.1 Purpose

The LOOP page eviction algorithm is the simplest form of the RANDOM page eviction strategy where the eviction candidates are selected round-robin based on their buffer frame index.

The LOOP eviction strategy uses a “pseudorandom” number generator—basically a *Counter-Based Random Number Generator* as in Subsection **??**, but instead of using a complex mapping function, the identity function modulo the number of buffer pool frames is used to generate the pseudorandom numbers— which generates an ordered sequence of buffer indexes.

3.2 Compared Counter Implementations

Six different counter implementations were evaluated for the use in a LOOP page eviction algorithm. They can be grouped as follows:

- blocking counters
  - mutex counter
  - spinlock counter
- non-blocking counters
  - modulo counter
  - lock-free counter
- local counters
  - local counter
  - local modulo counter

The counters return values from 1 to `blockCount` − 1—the range of the buffer frame indexes.

3.2.1 Mutex Counter

The *mutex counter* is a blocking counter which uses one global counter—used to select candidates for page eviction—synchronized using a mutex called through the `std::mutex` interface.

```
inline uint32_t mutexCounter() {
    static std::mutex indexLock;
    static uint32_t lastIndex = 0;
    std::lock_guard<std::mutex> guard(indexLock);
    lastIndex++;
    if (lastIndex >= blockCount) {
        lastIndex = 1;
    }
    return lastIndex;
}
```

3.2.2 Spinlock Counter

The *spinlock counter* is also a blocking counter using one global counter but it is protected by a spinlock implemented using a `std::atomic_flag`.

```
inline uint32_t spinlockCounter() {
    static std::atomic_flag indexLock =
        ATOMIC_FLAG_INIT;
    static uint32_t lastIndex = 0;
    uint32_t newIndex;
    while (indexLock.test_and_set(
                std::memory_order_acquire)) {}
    lastIndex++;
    if (lastIndex >= blockCount) {
        lastIndex = 1;
    }
    newIndex = lastIndex;
    indexLock.clear(std::memory_order_release);
    return newIndex;
}
```

3.2.3 Modulo Counter

The *modulo counter* is a non-blocking counter which uses atomic increment operations on a global counter (of type `std::atomic<uint64_t>`). The modulo is calculated thread-locally and therefore, the value of the global counter is strictly increasing.

```
inline uint32_t moduloCounter() {
    static std::atomic<uint64_t> lastIndex;
    while (true) {
        uint32_t newIndex = lastIndex++
                            % blockCount;
        if (newIndex != 0) {
            return newIndex;
        } else {
            continue;
        }
    }
}
```

3.2.4 Local Counter

The *local counter* is—obviously—a local counter where each thread performing page evictions uses its own circular counter.

```
inline uint32_t localCounter() {
    static thread_local uint32_t lastIndex = 0;
    lastIndex++;
    if (lastIndex >= blockCount) {
        lastIndex = 1;
    }
    return lastIndex;
}
```

3.2.5 Local Modulo Counter

The *local modulo counter* is a local counter where the circular counting is not achieved with a branch operation but with a possibly cheaper modulo operation calculated during each call.

```
inline uint32_t localModuloCounter() {
    static thread_local uint64_t lastIndex = 0;
    static uint32_t moduloDivisor = blockCount
                                    - 1;
    return (lastIndex++ % moduloDivisor) + 1;
}
```

3.2.6 Lock-Free Counter

The *lock-free counter* is a non-blocking counter which uses atomic increment operations on a global counter (of type `std::atomic<uint32_t>`) and algorithm-specific synchronization to achieve circular counting using branch operations.

```
inline uint32_t lockFreeCounter() {
    static std::atomic<uint32_t> newIndex(1);
    uint32_t pickedIndex = newIndex;
    if (pickedIndex < blockCount) {
        newIndex++;
        return pickedIndex;
    } else {
        return newIndex = 1;
    }
}
```

3.3 Performance Evaluation

Due to the implementation independence of the eviction decisions of LOOP page evictioners, the achieved hit rates achieved in an exemplary DBMS are not required as performance measure for this evaluation. The only performance difference between the different LOOP page evictioners is the overhead imposed by the concurrent counting. Therefore, a microbenchmark measuring only the execution time of the counting is appropriate.

The variables of the evaluation are the concurrent counter implementation—the alternatives presented in the previous section are evaluated—and the number of threads concurrently incrementing the counter. The smallest (> 0) and largest integers returned by the counters—representing the smallest and largest buffer pool indexes—do not significantly influence the performance of the LOOP page evictioners. Therefore, this integer interval [1..999] is a constant in this evaluation.

3.3.1 Microbenchmark

The microbenchmark used for the performance evaluation of the LOOP page eviction algorithms forks a given number of worker threads—each of them calls the evaluated concurrent counter a given number of times (1 000 000)—and measures the wall time elapsed until all the worker threads finished their operation and joined.

3.3.2 Configuration of the Used System A

- **CPU:** 2× *Intel® Xeon® Processor X5670* @6 × 2.93GHz from early 2010
- **Main Memory:** 12 × 8GB = 96GB of DDR2-SDRAM @1333MHz
- **OS:** *Ubuntu 18.04*

3.3.3 Configuration of the Used System B

- **CPU:** *Intel® Core™ i7-8700* @12 × 3.2 GHz from late 2017
- **Main Memory:** 2 × 8GB = 16GB of DDR4-SDRAM @2666 MHz
- **OS:** *Ubuntu 19.10*

3.3.4 Microbenchmark Results

Figure 3.1 shows the LOOP index generation performance of the NUMA system A—figure 3.2 shows the performance of the UMA system B. The LOOP index generation throughput is total number of indexes generated (from all working threads) per time.

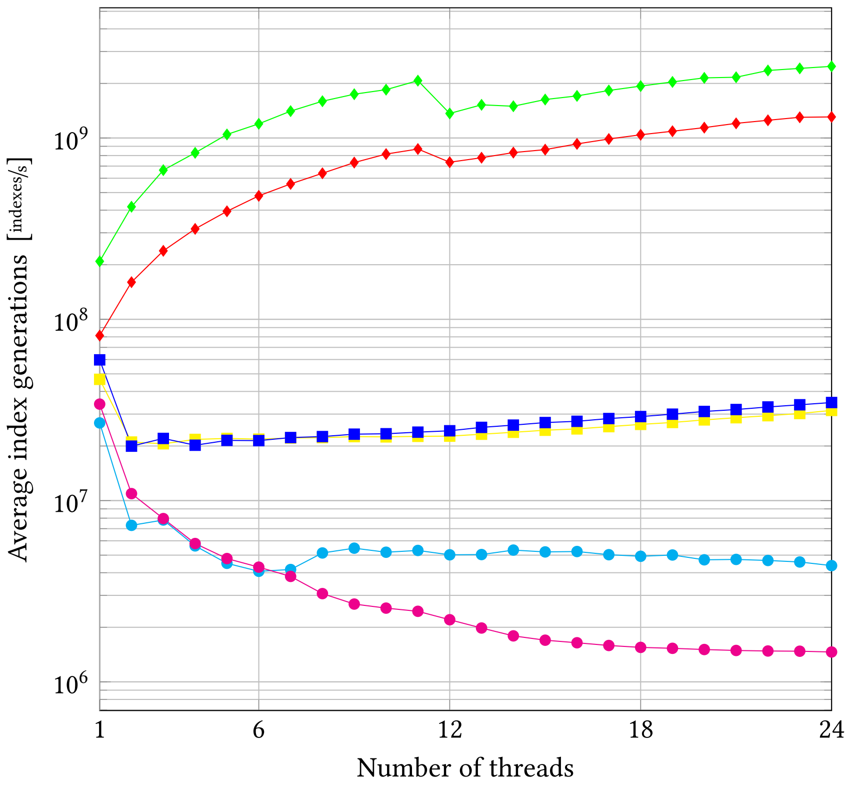

Figure 3.1: The throughput of index generations of the evaluated LOOP implementations on system A

The *blocking counters—mutex counter* —●— and *spinlock counter* —●——are the slowest counters on both systems. The locking overhead and—for higher numbers of working threads—the lock contention make these concurrent counters more than one order of magnitude slower than the *non-blocking counters*. The more optimized mutex, which uses low-overhead busy-waiting (spinlock behavior) in situations of low contention and higher-overhead queuing and context switches in situations of higher contention, can better utilize the available physical CPU cores by suspending waiting threads—and therefore, omitting unnecessary uses of hyper-threading.

The *non-blocking counters—modulo counter* —■— and *lock-free counter*

─■─—perform almost identical to each other. In situations without contention, the only serious overhead of those counters are the conditional branch operations. But most of the times, a modern CPU can correctly predict the targets of these branches. But when there are multiple working threads counting, the atomic operations on the counter variable are the major overhead slowing down the *non-blocking counters* compared to *local counters*.

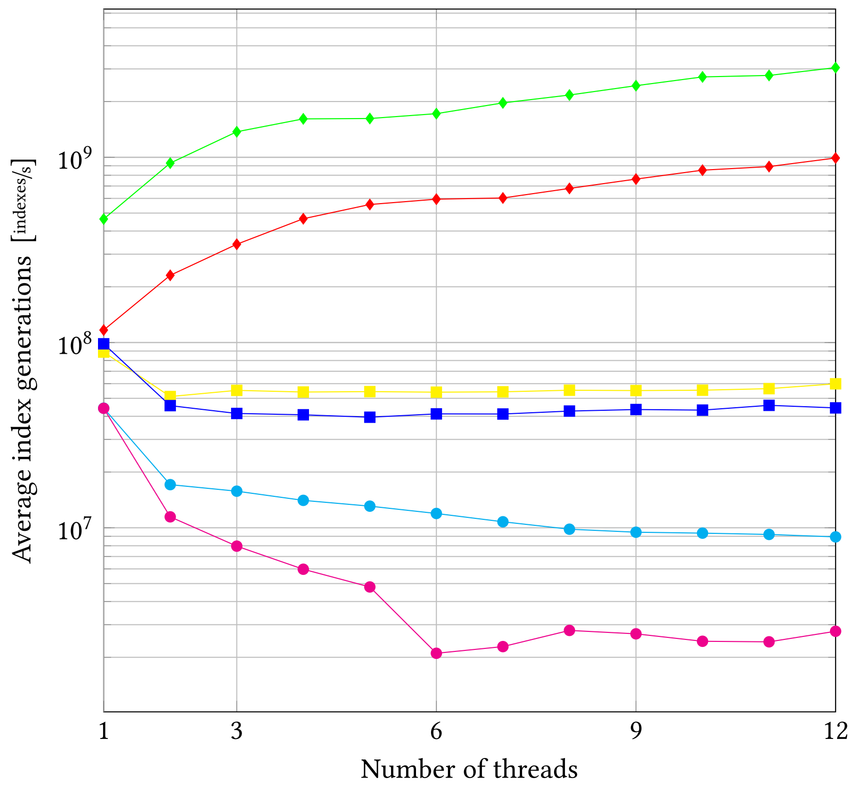

Figure 3.2: The throughput of index generations of the evaluated LOOP implementations on system B

Obviously, the *local counters*—*local counter* ─♦─ and *local modulo counter* ─♦─—are the fastest counters because they completely omit any synchronization between the counting threads. While the other counters count based on a global counter, these ones count based on local counters, resulting in a thread-wise round-robin selection of eviction candidates instead of a global round-robin selection of eviction candidates. The independence of the threads makes these counters very scalable as long as there are hardware threads available. This results in a performance advantage of up to almost two orders of magnitude compared to the *non-blocking counters*. The significantly higher performance of the *local counter* in comparison to the *local modulo counter* is the result of the slow modulo operation when the divisor is not a power of two. The branch target of the conditional branch in the *local counter* is predicted correctly most of the times, making its overhead negligible.

3.4 Conclusion

Each LOOP page eviction algorithm is a good replacement for any other RANDOM page eviction algorithm. Tests in an OLTP database showed a hit rate of the LOOP strategy not worse than the one of any RANDOM page eviction strategy and the overheads of the LOOP page eviction strategies—especially of the *local counters*—are lower than the ones of any other RANDOM page eviction strategy. This makes the LOOP page evictioners superior to the other RANDOM page evictioners in OLTP applications (represented by the TPC-C benchmark). Due to the fact, that the use of local counters instead of a global counter does not change the hit rate in the database, the *local counter*—the LOOP page eviction algorithm with the lowest overhead—can be recommended.

Bibliography

[AKY10] Yehuda Afek, Guy Korland, and Eitan Yanovsky. "Quasi-Linearizability: Relaxed Consistency for Improved Concurrency". In: *Principles of Distributed Systems.* Lecture Notes in Computer Science (6490 2010): *14th International Conference, OPODIS 2010 Tozeur, Tunisia, December 14-17, 2010 Proceedings.* Ed. by Chenyang Lu, Toshimitsu Masuzawa, and Mohamed Mosbah. Found. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen, pp. 395–410. DOI: 10.1007/978-3-642-17653-1_29. URL: https://bit.ly/2shYTDH (visited on Jan. 24, 2019).

[APD15] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. "Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data.* SIGMOD '15. Melbourne, Victoria, Australia: ACM, May 2015, pp. 707–722. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2749441. URL: https://bit.ly/2F9omlS (visited on Dec. 31, 2019).

[Bel66] Laszlo A. Belady. "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.

[Doh+04] Simon Doherty et al. "Formal Verification of a Practical Lock-Free Queue Algorithm". In: *Formal Techniques for Networked and Distributed Systems – FORTE 2004.* Lecture Notes in Computer Science (3235 2004): *24th IFIP WG 6.1 International Conference, Madrid Spain, September 27-30, 2004 Proceedings.* Ed. by David de Frutos-Escrig and Manuel Núñez. Found. by Gerhard

Goos, Juris Hartmanis, and Jan van Leeuwen, pp. 97–114. DOI: 10.1007/978-3-540-30232-2_7.

[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.

[Hen+10] Danny Hendler et al. "Flat Combining and the Synchronization-Parallelism Tradeoff". In: (2010): *SPAA '10: Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures.* Ed. by Friedhelm Meyer auf der Heide and Cynthia Phillips, pp. 355–364. DOI: 10.1145/1810479.1810540. URL: https://bit.ly/2YFIMfm (visited on Jan. 24, 2019).

[HSS07] Moshe Hoffman, Ori Shalev, and Nir Shavit. "The Baskets Queue". In: *Principles of Distributed Systems.* Lecture Notes in Computer Science (4878 2007): *11th International Conference, OPODIS 2007 Guadeloupe, French West Indies, December 17-20, 2007 Proceedings.* Ed. by Eduardo Tovar, Philippas Tsigas, and Hacène Fouchal. Found. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen, pp. 401–414. DOI: 10.1007/978-3-540-77096-1_29. URL: https://bit.ly/2EoTfmd (visited on Jan. 24, 2019).

[Jam94] Frederick E. James. "RANLUX: A Fortran Implementation of the High-Quality Pseudorandom Number Generator of Lüscher". In: *Computer Physics Communications* 79 (1 Feb. 1994), pp. 111–114. ISSN: 0010-4655. DOI: 10.1016/0010-4655(94)90233-X.

[LS04] Edya Ladan-Mozes and Nir Shavit. "An Optimistic Approach to Lock-Free FIFO Queues". In: *Distributed Computing.* Lecture Notes in Computer Science (3274 2004): *18th International Conference, DISC 2004, Amsterdam, The Netherlands, October 4-7, 2004 Proceedings.* Ed. by Rachid Guerraoui. Found. by Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen, pp. 117–

131. DOI: 10.1007/978-3-540-30186-8_9. URL: https://bit.ly/34dyfJy (visited on Jan. 24, 2019).

[Lüs94] Martin Lüscher. "A Portable High-Quality Random Number Generator for Lattice Field Theory Simulations". In: *Computer Physics Communications* 79 (1 Feb. 1994), pp. 100–110. ISSN: 0010-4655. DOI: 10.1016/0010-4655(94)90232-1.

[MN98] Makoto Matsumoto and Takuji Nishimura. "Mersenne Twister: A 623-Dimensionally Equidistributed Uniform Pseudo-Random Number Generator". In: *ACM Transactions on Modeling and Computer Simulation (TOMACS)* 8 (1 Jan. 1998): *Special Issue on Uniform Random Number Generation.* Ed. by Philip Heidelberger, Raymond Coutre, and Pierre L'Ecuyer, pp. 3–30. ISSN: 1049-3301. DOI: 10.1145/272991.272995. URL: https://bit.ly/2qQD0uU (visited on Jan. 24, 2019).

[MS96] Maged M. Michael and Michael L. Scott. "Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms". In: (1996): *PODC '96: Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing.* Ed. by James E. Burns and Yoram Moses, pp. 267–275. DOI: 10.1145/248052.24810629. URL: https://bit.ly/34cP9rz (visited on Jan. 24, 2019).

[PM88] Stephen K. Park and Keith W. Miller. "Random Number Generators: Good Ones Are Hard To Find". In: *Communications of the ACM* 31 (10 Oct. 1988), pp. 1192–1201. ISSN: 0001-0782. DOI: 10.1145/63039.63042. URL: https://bit.ly/2LKj6cf (visited on Jan. 30, 2019).

[PMS93] Stephen K. Park, Keith W. Miller, and Paul K. Stockmeyer. "Technical Correspondence: Response". In: *Communications of the ACM* 36 (7 July 1993): *Special Issue on Computer Augmented Environments: Back to the Real World,* pp. 108–110. ISSN: 0001-0782. DOI: 10.1145/159544.376068. URL: https://bit.ly/2PJ1PkT (visited on Jan. 30, 2019).