

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

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

This page intentionally left blank.

Contents

1	Buffer Frame Free List	1
1.1	Purpose	1
1.2	Compared Open Source Queue Implementations	1
1.2.1	Boost Lock-Free Queue with Variable Size	2
1.2.2	Boost Lock-Free Queue with Fixed Size	2
1.2.3	CDS Basket Lock-Free Queue	3
1.2.4	CDS Flat-Combining Sequential Queue	3
1.2.5	CDS Michael & Scott Lock-Free Queue	4
1.2.6	CDS Variation of Michael & Scott Lock-Free Queue	4
1.2.7	CDS Michael & Scott Blocking Queue with Fine-Grained Locking	4
1.2.8	CDS Ladan-Mozes & Shavit Optimistic Queue	5
1.2.9	CDS Segmented Queue	5
1.2.10	CDS Vyukov's MPMC Bounded Queue	5
1.2.11	Folly MPMC Queue	6
1.2.12	Dmitry Vyukov's Bounded MPMC Queue	6
1.2.13	Gavin Lambert's MPMC Bounded Lock-Free Queue	6
1.2.14	Matt Stump's Bounded MPMC Queue	6
1.2.15	Erik Rigtorp's Bounded MPMC Queue	7
1.2.16	TBB Concurrent Queue	7
1.2.17	TBB Bounded Concurrent Dual Queue	7
1.3	Performance Evaluation	8
1.3.1	Microbenchmark	8
1.3.2	Used Versions of the Libraries and Queue Implementations	8
1.3.3	Configuration of the Used System	9
1.3.4	Microbenchmark Results	9
1.4	Conclusion	11

2	RANDOM Page Eviction	12
2.1	Purpose	12
2.2	Compared Pseudorandom Number Generators	12
2.2.1	Linear Congruential Generator (LCG) – 1958 . . .	13
2.2.1.1	Lehmer Generator – 1949	13
2.2.1.2	Park-Miller Generator – 1988	14
2.2.1.3	MIXMAX Generator – 1991	14
2.2.1.4	Permuted Congruential Generator (PCG) – 2014	15
2.2.2	Lagged Fibonacci Generator (LFG) – 1958	16
2.2.2.1	Subtract-With-Borrow (SWB) – 1991 . .	16
2.2.3	Linear Feedback Shift Register (LFSR) – 1965 . . .	17
2.2.3.1	Mersenne Twister (MT) – 1998	18
2.2.3.2	Xorshift – 2003	19
2.2.3.3	Well Equidistributed Long-Period Linear (WELL) – 2006	19
2.2.3.4	Xoshiro – 2018	21
2.2.3.5	Xoroshiro – 2018	22
2.2.4	Inversive Congruential Generator (ICG) – 1986 . .	23
2.2.5	ranshi – 1995	23
2.2.6	Gjrand – 2005	24
2.2.7	A Small Noncryptographic PRNG (JSF) – 2007 . . .	24
2.2.8	SFC – 2010	25
2.2.9	Counter-Based Random Number Generator (CBRNG) – 2011	25
2.2.9.1	ARC4 – 1997	25
2.2.9.2	ChaCha – 2008	26
2.2.9.3	Advanced Randomization System (ARS) – 2011	26
2.2.9.4	Threefry – 2011	26
2.2.9.5	Philox – 2011	27
2.2.9.6	Advanced Encryption Standard (AES) – 2011	27
2.2.10	SplitMix – 2014	27
2.2.11	Combinations of different PRNG	28
2.2.12	Biased Uniform Integer Distribution	28

Contents

2.3	Performance Evaluation	30
2.3.1	Microbenchmark	33
2.3.2	Configuration of the Used System	33
2.3.3	Benchmark Results	33
2.4	Conclusion	33
3	LOOP Page Eviction	35
3.1	Purpose	35
3.2	Compared Counter Implementations	35
3.2.1	Mutex Counter	36
3.2.2	Spinlock Counter	36
3.2.3	Modulo Counter	37
3.2.4	Local Counter	37
3.2.5	Local Modulo Counter	37
3.2.6	Lock-Free Counter	38
3.3	Performance Evaluation	38
3.3.1	Microbenchmark	39
3.3.2	Configuration of the Used System	39
3.3.3	Microbenchmark Results	39
3.4	Conclusion	41
	Bibliography	42

This page intentionally left blank.

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 queue² 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++ library⁴ 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 combining⁶. 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/2ZEsY0>

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 algorithm⁹, 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 algorithm¹⁰ 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 implementation¹¹ 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 at 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>

1.2 Compared Open Source Queue Implementations

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 implementation¹² 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 implementation¹³ 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 enqueueing 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 implementation¹⁴ provided by the *Concurrent Data Structures* C++ library is **bounded** and was developed by D. Vyukov¹⁵.

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

This¹⁷ 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

This¹⁸ 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

This¹⁹ 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/35a51KL>

¹⁸<https://bit.ly/2F7jv1b>

¹⁹<https://github.com/mstump/queues>

1.2.15 Erik Rigtorp's Bounded MPMC Queue

The **bounded lock-free** queue²⁰ 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* library²¹ is an open source library originally developed by Intel®. The first thread-safe queue implementation²² 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, enqueueing buffer frame indexes of emptied buffer frames.

1.2.17 TBB Bounded Concurrent Dual Queue

The other thread-safe queue implementation²³ 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 enqueueing 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 2.3.1²⁴
- *Folly* a8d1fd8²⁵
- Dmitry Vyukov's Bounded MPMC Queue as of September 2017²⁶

²⁴<https://bit.ly/39vysvB>

²⁵<https://bit.ly/2sy4J41>

²⁶<https://bit.ly/2MHbXtG>

1.3 Performance Evaluation

- Gavin Lambert’s MPMC Bounded Lock-Free Queue [e409068](https://bit.ly/2sy6QoN)²⁷
- Matt Stump’s MPMC Queue [319c253](https://bit.ly/2ZBEMgk)²⁸
- Erik Rigtorp’s MPMC Queue [553cf42](https://bit.ly/2F7cH7d)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>

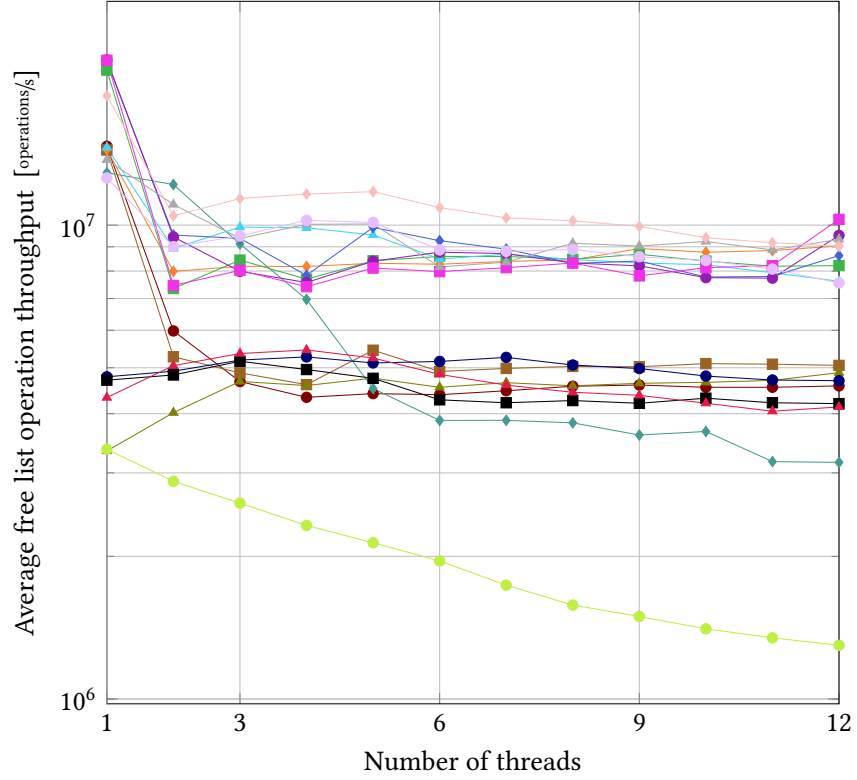


Figure 1.1: The operation throughput of the evaluated free list queue implementations

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*

1.4 Conclusion

—■— 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 evictionner—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 evictionner to decide on the page to evict 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—appropriate for cryptographic applications—others take only few CPU cycles and almost no memory to generate a random number.

2.2 Compared Pseudorandom Number Generators

Due to the enormous number of PRNG described in literature, an exhaustive comparison of PRNG for the use in RANDOM page evictioners is not possible in this context. Therefore, only a small selection of PRNG—mostly from the *C++ Standard Library*¹ and the *Boost Random Number Library*² (part of the *Boost C++ Libraries*³)—were selected for this evaluation.

2.2.1 Linear Congruential Generator (LCG) – 1958

The *linear congruential generator*—a generalization of the earlier proposed *Lehmer generator*—is a family of PRNG that was independently proposed by W. E. Thomson in [Tho58] and by A. Rotenberg in [Rot60].

A LCG is defined by the following recurrence relation X :

$$X_{n+1} = (a \cdot X_n + c) \bmod m \quad n \geq 0$$

In this definition, $a \in (0, m)$ is the multiplier, $c \in [0, m)$ is the increment, $m \in (0, \infty)$ is the modulus and $X_0 \in [0, m)$ is the seed.

The following members of the LCG family of PRNG that are not members of specializations defined in subsections were compared:

- **rand**: $a = 0x41C64E6D$, $c = 0x3039$, $m = 2^{31}$ if using *GNU C Library*⁴
- **rand48**: $a = 0x5DEECE66D$, $c = 0xB$, $m = 2^{48}$
- **Kreutzer1986**: Buffers 97 random numbers of a LCG with $a = 0x5556$, $c = 0x24D69$, $m = 0xAE529$ and returns them shuffled according to an algorithm proposed by Carter Bays and S. D. Durham in [BD76]. This was proposed by Wolfgang Kreutzer in [Kre86].

2.2.1.1 Lehmer Generator – 1949

The *Lehmer generator* (also known as *multiplicative congruential generator*) is the earliest family of PRNG of “usable” quality, proposed by Derrick H. Lehmer in [Leh51] in 1949.

¹<https://en.cppreference.com/w/cpp>

²https://www.boost.org/doc/libs/release/doc/html/boost_random.html

³<https://www.boost.org/>

⁴<https://www.gnu.org/software/libc/>

It is a specialization of the later proposed LCG with $c = 0$.

The following members of the MCG family of PRNG were compared:

- **MCG128**: $a = 0x1168C7BF168D765C661FD0407A968ADD$, $m = 2^{64} - 1$, 128 bit state
- **MCG128Fast**: MCG128 with $a = 0xDA942042E4DD58B5$
- **RANECU**: Combination of two Lehmer generators ($a_1 = 0x9C4E$, $m_1 = 0x7FFFFFFAB$, $a_2 = 0x9EF4$, $m_2 = 0x7FFFFFF07$) where the output is $o_1 - o_2$ if $o_2 < o_1$ or $o_1 - o_2 + 0x7FFFFFFAA$ (unsigned 32 bit output) else for o_1, o_2 random numbers generated by the two Lehmer generators. This was proposed by Pierre L'Ecuyer in [LEc88] and modified by F. James in [Jam90].

2.2.1.2 Park-Miller Generator – 1988

The *Park-Miller generator* (now known as MINSTD) is a set of parameters for the *Lehmer generator* proposed by Stephen K. Mark and Keith W. Miller in [PM88]. After criticism by George Marsaglia and Stephen Sullivan, they proposed a changed set of parameters in [PMS93].

In their initial proposal, the parameters were $a = 16807$ and $m = 2^{31} - 1$. In their later proposal, they used $a = 48271$ instead.

The following Park-Miller generators were compared:

- **MINSTD0**: $a = 0x41A7$, $m = 2^{31} - 1$
- **MINSTD**: $a = 0xBC8F$, $m = 2^{31} - 1$
- **KnuthB**: Buffers 256 random numbers of MINSTD0 and returns them shuffled according to an algorithm proposed by Carter Bays and S. D. Durham in [BD76]. This was proposed by Donald E. Knuth in [Knu81].

2.2.1.3 MIXMAX Generator – 1991

The *MIXMAX generator* is a *matrix linear congruential generator* proposed by G. K. Savvidy and N. G. Ter-Arutyunyan-Savvidy in [ST91].

2.2 Compared Pseudorandom Number Generators

In contrast to a LCG, a matrix LCG uses a $N \times N$ -matrix of multipliers A instead of a multiplier a .

$$a'_i = \begin{cases} \left(\sum_{j=1}^N A_{ij} \cdot a_j \right) \bmod m + s \cdot a_2 & \text{if } i = 3 \\ \left(\sum_{j=1}^N A_{ij} \cdot a_j \right) \bmod m & \text{else} \end{cases}$$

In this definition, $s \in \mathbb{Z}$ is a small “magic” integer, $m \in (0, \infty)$ is the modulus and the initial N -dimensional vector a is the seed.

The following members of the MIXMAX family of PRNG was compared:

- **MixMax2.0:** $N = 17$, $s = 0$, $m = 2^{36} + 1$

2.2.1.4 Permuted Congruential Generator (PCG) – 2014

The *permuted congruential generator* is a modified *linear congruential generator* proposed by Melissa E. O’Neill in [ONe14].

In contrast to a typical LCG, the PCG state has double the width of its output, the modulus m is $m = 2^k$ for $k \in \mathbb{N}$ and the output is generated by a state-defined bit-wise rotation of the state.

The following members of the PCG family of PRNG were compared:

- **PCG32:** $a = 0x5851F42D4C957F2D$, $c = 0x14057B7EF767814F$, $m = 2^{31} - 1$, 64 bit state
- **PCG32Unique:** *PCG32* where c is based on a memory address
- **PCG32Fast:** *PCG32* where $c = 0$
- **PCG32K2:** 2-dimensionally equidistributed version of *PCG32*
- **PCG32K2Fast:** 2-dim. equidistributed version of *PCG32Fast*
- **PCG32K64:** 64-dim. equidistributed version of *PCG32*
- **PCG32K64Fast:** 64-dim. equidistributed version of *PCG32Fast*
- **PCG32K1024:** 1024-dim. equidistributed version of *PCG32*
- **PCG32K1024Fast:** 1024-dim. equidistributed version of *PCG32Fast*
- **PCG32K16384:** 16384-dim. equidistributed version of *PCG32*
- **PCG32K16384Fast:** 16384-dim. equidistributed version of *PCG32Fast*

2.2.2 Lagged Fibonacci Generator (LFG) – 1958

The *lagged Fibonacci generator* is a family of PRNG—based on the generalization of the Fibonacci sequence—proposed (but never published) by G. J. Mitchell and D. P. Moore in 1958.

A LFG is defined by the following recurrence relation X :

$$X_n = (X_{n-j} + X_{n-k}) \mod m, n \geq j \wedge n \geq k$$

In this definition, $j = 24$ and $k = 55$ are the lags of the original proposal and $(X_0, \dots, X_{\max(j,k)})$ is the seed to be seeded based on e.g. another random number generator.

The following (floating-point) members of the LFG family of PRNG that are not members of specializations defined in subsections were compared:

- **LaggedFibonacci607**: $j = 607, k = 273, m = 1$
- **LaggedFibonacci1279**: $j = 1279, k = 418, m = 1$
- **LaggedFibonacci2281**: $j = 2281, k = 1252, m = 1$
- **LaggedFibonacci3217**: $j = 3217, k = 576, m = 1$
- **LaggedFibonacci4423**: $j = 4423, k = 2098, m = 1$
- **LaggedFibonacci9689**: $j = 9689, k = 5502, m = 1$
- **LaggedFibonacci19937**: $j = 19937, k = 9842, m = 1$
- **LaggedFibonacci23209**: $j = 23209, k = 13470, m = 1$
- **LaggedFibonacci44497**: $j = 44497, k = 21034, m = 1$
- **RANMAR**: $X_n = \begin{cases} X_{n-97} - X_{n-33} & \text{if } X_{n-97} \geq X_{n-33} \\ X_{n-97} - X_{n-33} + 1 & \text{else} \end{cases} \mod 1$
combined with a simple arithmetic sequence as proposed by G. Marsaglia et al. in [MZT90] and modified by F. James in [Jam90]

Many used parameters (lags) were proposed by R. P. Brent in [Bre92].

2.2.2.1 Subtract-With-Borrow (SWB) – 1991

The *subtract-with-borrow* generator is a modification of the *lagged Fibonacci generator* proposed by George Marsaglia and Arif Zaman in [MZ91].

2.2 Compared Pseudorandom Number Generators

A SWB generator is defined by the following iterating function f :

$$f(x_1, \dots, x_j, c) = \begin{cases} (x_{j+1-k} - x_1 - c, 0) & \text{if } x_{j+1-k} - x_1 - c \geq 0 \\ (x_{j+1-k} - x_1 - c + b, 1) & \text{if } x_{j+1-k} - x_1 - c < 0 \end{cases}$$

In this definition, $X_n = f(X_n)$ is the generated sequence. The lags j, k and the base b need to be chosen appropriately with $j > k$ and the initial seed vector (x_1, \dots, x_j, c) needs to be seeded based on e.g. another random number generator.

The following members of the SWB family of PRNG were compared:

- **Ranlux24Base**: $j = 24, k = 10, b = 2^{24} - 1$
- **Ranlux24**: *Ranlux24Base* discarding 200 per 223 generated numbers
- **Ranlux3**: *Ranlux24Base* discarding 199 per 223 generated numbers
- **Ranlux4**: *Ranlux24Base* discarding 365 per 389 generated numbers
- **Ranlux48Base**: $j = 12, k = 5, b = 2^{48} - 1$
- **Ranlux48**: *Ranlux48Base* discarding 378 per 389 generated numbers
- **Ranlux64_3**: $j = 10, k = 24, b = 2^{48} - 1$ discarding 199 per 223 generated numbers
- **Ranlux64_4**: $j = 10, k = 24, b = 2^{48} - 1$ discarding 365 per 389 generated numbers
- **Ranlux3_01**: floating-point version of *Ranlux3*
- **Ranlux4_01**: floating-point version of *Ranlux4*
- **Ranlux64_3_01**: floating-point version of *Ranlux64_3*
- **Ranlux64_4_01**: floating-point version of *Ranlux64_4*

The *Ranlux* family of SWB PRNG was proposed by M. Lüscher in [Lüs94].

2.2.3 Linear Feedback Shift Register (LFSR) – 1965

The *linear feedback shift register* PRNG is a family of PRNG proposed by Robert C. Tausworthe in [Tau65].

LSFR generators work on a bit sequence $a = \{a_k\}$ which is defined as follows:

$$a_k = c_1 \cdot a_{k-1} + c_2 \cdot a_{k-2} + \dots + c_n \cdot a_{k-n} \mod 2$$

The parameters $c_i \in \{0, 1\}$ with $1 \leq i \leq n$ are fixed and n is the bit width of the state. Based on this state, the random number y_k is generated as follows:

$$y_k = \sum_{t=1}^L 2^{-t} \cdot a_{qk+r-t}$$

Here, $L \leq n$ represents the bit width of the output random number, q is the number of bit between two successive y_k in a_k ($q \geq L$) and r is a random number in the state range $[0, 2^n - 1]$.

In [LEc96], Pierre L'Ecuyer proposed a specific PRNG as the combination of three LSFR generators using bitwise XOR operations. This LFSR PRNG was compared:

- **Taus88:** $n_1 = 32, \quad c_1 = 2^{32} - 2, \quad L_1 = 32, \quad q_1 = 12, \quad r_1 = 18$
 $n_2 = 32, \quad c_2 = 2^{32} - 2, \quad L_2 = 32, \quad q_2 = 4, \quad r_2 = 27$
 $n_3 = 32, \quad c_3 = 2^{32} - 2, \quad L_3 = 32, \quad q_3 = 17, \quad r_3 = 25$
- **Hurd160:** LFSR with 32 5 bit shift registers by W. J. Hurd in [Hur74]
- **Hurd288:** LFSR with 32 9 bit shift registers by W. J. Hurd in [Hur74]

2.2.3.1 Mersenne Twister (MT) – 1998

The *Mersenne Twister*—a twisted *generalized feedback shift register* (GFSR) operating on a state matrix—was proposed by M. Matsumoto and T. Nishimura in [MN98]. It is by far the most commonly used general-purpose PRNG.

A more detailed description of the design and internals of the MT is unfortunately beyond the scope of this thesis.

The following Mersenne Twisters—all proposed in the initial proposal of MT [MN98]—were compared:

- **MT19937:** Mersenne prime is $2^{19937} - 1$
- **MT19937-64:** Mersenne prime is $2^{19937} - 1$, 64 bit version
- **MT11213B:** Mersenne prime is $2^{11213} - 1$

2.2 Compared Pseudorandom Number Generators

2.2.3.2 Xorshift – 2003

The *xorshift*—a sub-type of the *linear feedback shift register* implemented purely using fast bitwise XOR and shift operations—was proposed by George Marsaglia in [Mar03].

The following implementation of a 32 bit *xorshift* was given in [Mar03]:

```
uint32_t xorshift32() {  
    static uint32_t state = 2463534242;  
    state ^= (state << 13);  
    state = (state >> 17);  
    return (state ^= (state << 5));  
}
```

The initial *state*—hard-coded in this example to 2463534242—should be randomly seeded in any real use case.

The following xorshift generators were compared:

- **xorshift32**: 32 bit xorshift
- **xorshift64***: 64 bit xorshift with truncated output
- **xorwow**: 128 bit xorshift combined with a Weyl sequence
- **xorshift128+**: 128 bit xorshift with 64 bit shifts ([Vig17])

2.2.3.3 Well Equidistributed Long-Period Linear (WELL) – 2006

The *well equidistributed long-period linear* generators—a family of PRNG of the form of GFSR and MT generators—was proposed by François Panneton et al. in [PLM06].

The WELL algorithm is as follows:

$$\begin{aligned} z_0 &\leftarrow (m_p \wedge v_{i,r-1}) \oplus (\tilde{m}_p \wedge v_{i,r-2}) \\ z_1 &\leftarrow T_0 \cdot v_{i,0} \oplus T_1 \cdot v_{i,m_1} \\ z_2 &\leftarrow T_2 \cdot v_{i,m_2} \oplus T_3 \cdot v_{i,m_3} \\ z_3 &\leftarrow z_1 \oplus z_2 \\ z_4 &\leftarrow T_4 \cdot z_0 \oplus T_5 \cdot z_1 \oplus T_6 \cdot z_2 \oplus T_7 \cdot z_3 \\ v_{i+1,r-1} &\leftarrow v_{i,r-2} \wedge m_p \\ \textbf{for } j &\leftarrow r - 2, 2 \textbf{ do} \end{aligned}$$

```

     $v_{i+1,j} \leftarrow v_{i,j-1}$ 
end for
 $v_{i+1,1} \leftarrow z_3$ 
 $v_{i+1,0} \leftarrow z_4$ 
return  $y_i = v_{i,0}$ 

```

In the algorithm, w is the bit-width of the random numbers output by the WELL algorithm, $r \in (0, \infty)$ and $p \in [0, w)$ are unique integers and $m_p \in (0, r)$ are bitmasks. The bit-width of the elements of the r -dimensional state vector x_i is w and the last p bits of the last element of this vector are 0. Possible values for the transformation $w \times w$ -matrices T_0, \dots, T_7 and further limitations to the parameters are given in [PLM06].

Shin Harase proposed a tempering method in [Har09] to make some WELL generators maximally equidistributed.

The following WELL generators were compared:

- **WELL512**: $w = 32, r = 16, p = 0, m_1 = 13, m_2 = 9, m_3 = 5$
- **WELL521**: $w = 32, r = 17, p = 23, m_1 = 13, m_2 = 11, m_3 = 10$
- **WELL607**: $w = 32, r = 19, p = 1, m_1 = 16, m_2 = 15, m_3 = 14$
- **WELL800**: $w = 32, r = 25, p = 0, m_1 = 14, m_2 = 18, m_3 = 17$
- **WELL1024**: $w = 32, r = 32, p = 0, m_1 = 3, m_2 = 24, m_3 = 10$
- **WELL19937**: $w = 32, r = 624, p = 31, m_1 = 70, m_2 = 179, m_3 = 449$
- **WELL21701**: $w = 32, r = 679, p = 27, m_1 = 151, m_2 = 327, m_3 = 84$
- **WELL23209**: $w = 32, r = 726, p = 23, m_1 = 667, m_2 = 43, m_3 = 462$
- **WELL44497**: $w = 32, r = 1391, p = 15, m_1 = 23, m_2 = 481, m_3 = 229$
- **WELL800-ME**: *WELL800* with $y_i \leftarrow v_{i,0} \oplus (v_{i,19} \wedge 0x4880)$
- **WELL19937-ME**: *WELL19937* with $y_i \leftarrow v_{i,0} \oplus (v_{i,180} \wedge 0x4118000)$
- **WELL21701-ME**: *WELL21701* with $y_i \leftarrow v_{i,0} \oplus (v_{i,328} \wedge 0x1002)$
- **WELL23209-ME**: *WELL23209* with $y_i \leftarrow v_{i,0} \oplus (v_{i,44} \wedge 0x5100000)$
- **WELL44497-ME**: *WELL44497* with $y_i \leftarrow v_{i,0} \oplus (v_{i,482} \wedge 0x48000000)$

2.2 Compared Pseudorandom Number Generators

2.2.3.4 Xoshiro – 2018

The *xoshiro*—a *linear feedback shift register* generator implemented using XOR, shift and rotate operations—was—together with *xoroshiro*—proposed by David Blackman and Sebastiano Vigna in [BV18].

The following (slightly modified) implementation of a 32 bit *xoshiro* with a 128 bit state was given in [BV18]:

```
void xoshiro128() {
    static uint32_t s0_ = 0x01d353e5f3993bb1;
    static uint32_t s1_ = 0xf7381bed96327640;
    static uint32_t s2_ = 0xfdfcaa91110765b5;
    static uint32_t s3_ = 0x0;
    const uint64_t t = s1_ << a;
    s2_ ^= s0_;
    s3_ ^= s1_;
    s1_ ^= s2_;
    s0_ ^= s3_;
    s2_ ^= t;
    s3_ = (s3_ << b) | (s3_ >> (32 - b));
}
```

The initial `s0_`, `s1_`, `s2_` and `s3_` which are the state—hard-coded in this example to `0x01d353e5f3993bb1`, `0xf7381bed96327640`, `0xfdfcaa91110765b5` and `0x0`—should be randomly seeded in any real use case. For the 32 bit case, the authors proposed shift and rotate values to be `a = 9` and `b = 11`.

It can be easily seen in the implementation, that *xoshiro* does not define the generation of a pseudorandom number from its state. The authors proposed four scramblers to be used with *xoshiro* (and *xoroshiro*) where the two more advanced ones try to eliminate linear artifacts from the state.

- + **scrambler** The simple + scrambler returns just the sum of two of the state words (e.g. **return** `s0_ + s3_`).
- * **scrambler** The not any less simple * scrambler returns just the product of one of the state words with a fixed, odd multiplier (e.g. **return** `s1_ + mult`).
- ++ **scrambler** The ++ scrambler first adds up two of the state words, rotates the sum to the left by `r` positions and returns the sum of this rotated

sum and the first of the two state words used in the first sum (e.g. **return** `((s0_+s3_) << r) | ((s0_+s3_) >> (32-r)) + s0_`). The authors propose $r = 7$ in the 32 bit case.

**** scrambler** The **** scrambler** first multiplies one of the state words with a fixed, odd multiplier s , rotates the product to the left by r positions and returns the product of this rotated product and another fixed, odd multiplier t (e.g. **return** `((s1_*s) << r) | ((s1_*s) >> (32-r)) * t`). The authors propose $s = 5$, $r = 7$ and $t = 9$ in the 32 bit case.

The following xoshiro generators were compared:

- **xoshiro128+32**: 32 bit *xoshiro* with 128 bit state and + scrambler
- **xoshiro128**32**: 32 bit *xoshiro* with 128 bit state and ** scrambler

2.2.3.5 Xoroshiro – 2018

The *xoroshiro*—another *linear feedback shift register* generator implemented using XOR, shift and rotate operations—was proposed by David Blackman and Sebastiano Vigna in [BV18].

The following (slightly modified) implementation of a 64 bit *xoroshiro* with a 128 bit state was given in [BV18]:

```
void xoroshiro128() {
    static uint64_t s0_ = 0xc1f651c67c62c6e0;
    static uint64_t s1_ = 0x30d89576f866ac9f;
    const uint64_t t = s0_ ^ s1_;
    s0_ = ((s0_ << a) | (s0_ >> (64 - a)))
        ^ t ^ (t << b);
    s1_ = (t << c) | (t >> (64 - c));
}
```

The initial $s0_$ and $s1_$ which are the state—hard-coded in this example to `0xc1f651c67c62c6e0` and `0x30d89576f866ac9f`—should be randomly seeded in any real use case. For the 64 bit case, the authors proposed shift and rotate values to be $a = 24$, $b = 16$ and $c = 37$.

2.2 Compared Pseudorandom Number Generators

For the generation of pseudorandom numbers from the states of a *xoroshiro* generator, the scramblers +, *, ++ and **, that are also used for *xoshiro*, are used. The details of these scramblers are described in Subsection 2.2.3.4.

The following xoroshiro generators were compared:

- **xoroshiro128+32**: 64 bit *xoroshiro* with 128 bit state and + scrambler
- **xoroshiro64+32**: 32 bit *xoroshiro* with 64 bit state and + scrambler
- **xoroshiro64*32**: 32 bit *xoroshiro* with 64 bit state and * scrambler
- **xoroshiro64**32**: 32 bit *xoroshiro* with 64 bit state and ** scrambler

2.2.4 Inversive Congruential Generator (ICG) – 1986

The *inversive congruential generator* is a family of PRNG proposed by Jürgen Eichenauer and Jürgen Lehn in [EL86].

A ICG is defined by the following recurrence relation X :

$$X_{n+1} = \begin{cases} (a \cdot X_n^{-1} + b) \bmod p & \text{if } X_n \neq 0 \\ b & \text{else} \end{cases}$$

In this definition, $a \in \mathbb{N}$ is the multiplier, $b \in \mathbb{N}$ is the increment, p is the prime modulus and $X_0 \in [0, p)$ is the seed. X_n^{-1} is the multiplicative inverse of X_n in the finite field $GF(p)$.

The following member of the ICG family of PRNG was compared:

- **Hellekalek1995**: $a = 0x238E$, $b = 0x7DCD313A$, $p = 0x7FFFFFFF$ as proposed by Peter Hellekalek in [Hel95]

2.2.5 ranshi – 1995

The *ranshi* algorithm is a PRNG proposed by F. Gutbrod in [Gut95].

The idea behind the algorithm is a physical system made of a number of black balls each with a position and a spin (state of the PRNG). A red ball—having also a spin and a position—colliding with the black balls is used to generate pseudorandom numbers.

2.2.6 Gjrand – 2005

The *gjrand* algorithm is based on a random invertible mapping⁵ of addition, XOR and rotate operations. This family of PRNG was proposed by David Blackman⁶.

The following member of the *gjrand* family of PRNG was compared:

- **gjrand32**: 32 bit PRNG with 128 bit state, parameters from the author

2.2.7 A Small Noncryptographic PRNG (JSF) – 2007

The *JSF* algorithm is based on a reversible, nonlinear function where all internal state bits affect one another of addition, XOR, rotate and conditional branch operations. This family of PRNG was proposed by Bob Jenkins⁷.

The following implementation (with $b_ = c_ = d_$ properly seeded) of a 32 bit *JSF* was used:

```
uint32_t jsf32() {
    static uint32_t a_ = 0xf1ea5eed;
    static uint32_t b_ = 0xcafe5eed00000001;
    static uint32_t c_ = 0xcafe5eed00000001;
    static uint32_t d_ = 0xcafe5eed00000001;
    uint32_t e = a_ - ((b_ << p)
                      | (b_ >> (32 - p)));
    a_ = b_ ^ ((c_ << q) | (c_ >> (32 - q)));
    b_ = c_ + (r ? ((d_ << r)
                  | (d_ >> (32 - r))) : d_);
    c_ = d_ + e;
    d_ = e + a_;
    return d_;
}
```

The following members of the *JSF* family of PRNG were compared:

- **JSF32n**: $p = 27, q = 17, r = 0$
- **JSF32r**: $p = 23, q = 16, r = 11$

⁵<http://www.pcg-random.org/posts/random-invertible-mapping-statistics.html>

⁶<http://gjrand.sourceforge.net/>

⁷<http://burtleburtle.net/bob/rand/smallprng.html>

2.2.8 SFC – 2010

The *SFC* algorithm is based on a random invertible mapping⁵ of addition, XOR, shift and rotate operations. This family of PRNG was proposed by Chris Doty-Humphrey as part of his PractRand⁸ statistical test and PRNG library.

The following implementation (with *a_*, *b_* and *c_* properly seeded) of a 32 bit *SFC* was used:

```
uint32_t sfc32() {
    static uint32_t a_ = 0xcafef00dbeef5eed;
    static uint32_t b_ = 0xcafef00dbeef5eed;
    static uint32_t c_ = 0xcafef00dbeef5eed;
    static uint32_t d_ = 0x1;
    uint32_t t = a_ + b_ + d_++;
    a_ = b_ ^ (b_ >> q);
    b_ = c_ + (c_ << r);
    c_ = (c_ << p) | (c_ >> (64 - p)) + t;
    return t;
}
```

The following member of the SFC family of PRNG was compared:

- **SFC32:** $p = 21, q = 9, r = 3$

2.2.9 Counter-Based Random Number Generator (CBRNG) – 2011

The *counter-based random number generator* is a family of PRNG proposed by J. Salmon et al. in [Sal+11].

The state of a CBRNG is a simple integer counter but the output mapping is done using a complex function—usually a cryptographic block cipher.

2.2.9.1 ARC4 – 1997

The *ARC4* is a PRNG first implemented in OpenBSD 2.1⁹ in 1997 for function `arc4random`.

⁸<http://pracrand.sourceforge.net/>

⁹<https://man.openbsd.org/arc4random>

It generates pseudorandom numbers from the keystream of the RC4 stream cipher which was released by Ronald L. Rivest in 1987. *ARC4* is not exactly a *CBRNG* because it uses a second state which is not a counter but the PRNG is closely related to the other *CBRNG* as it uses just a stream cipher to generate pseudorandom numbers.

2.2.9.2 ChaCha – 2008

ChaCha is a stream cipher proposed by Daniel J. Bernstein in [Ber08]. It is used as a PRNG by encoding the state of the PRNG—a simple integer counter—using the *ChaCha* stream cipher.

The following PRNG based on the family of ChaCha stream ciphers were compared:

- **ChaCha4:** Based on ChaCha 4-round cipher
- **ChaCha5:** Based on ChaCha 5-round cipher
- **ChaCha6:** Based on ChaCha 6-round cipher
- **ChaCha8:** Based on ChaCha 8-round cipher
- **ChaCha20:** Based on ChaCha 20-round cipher

2.2.9.3 Advanced Randomization System (ARS) – 2011

The *advanced randomization system* is a *counter-based random number generator* where the state—a simple integer counter—is mapped to the random output using a simplified AES block cipher.

The following ARS generator was compared:

- **ARS4x32:** 7 rounds, operating on four 32 bit integers

2.2.9.4 Threefry – 2011

The *Threefry* is a *counter-based random number generator* where the state is mapped to the random output using a simplified Threefish block cipher.

The following Threefry generators were compared:

- **Threefry2x32:** 20 rounds, operating on two 32 bit integers

2.2 Compared Pseudorandom Number Generators

- **Threefry4x32**: 20 rounds, operating on four 32 bit integers
- **Threefry2x64**: 20 rounds, operating on two 64 bit integers
- **Threefry4x64**: 20 rounds, operating on four 64 bit integers

2.2.9.5 Philox – 2011

The *Philox* is a *counter-based random number generator* where the state is mapped to the random output using a modified and simplified Threefish block cipher.

The following Threefry generators were compared:

- **Philox2x32**: 10 rounds, operating on two 32 bit integers
- **Philox4x32**: 10 rounds, operating on four 32 bit integers
- **Philox2x64**: 10 rounds, operating on two 64 bit integers
- **Philox4x64**: 10 rounds, operating on four 64 bit integers

2.2.9.6 Advanced Encryption Standard (AES) – 2011

The *Advanced Encryption Standard* PRNG is a *counter-based random number generator* where the state is mapped to the random output using the AES block cipher.

The following AES generator was compared:

- **AES4x32**: 10 rounds, operating on four 32 bit integers

2.2.10 SplitMix – 2014

SplitMix is a PRNG similar to the *CBRNGs* that was proposed by Guy Steele et al. in [SLF14]. It is derived from the PRNG *DotMix* which was proposed by Charles Leiserson et al. in [LSS12].

While the state of *CBRNGs* is advanced by adding 1—it is a simple counter—the state of *SplitMix* is advanced by adding a fixed γ . Instead of using a complex hash function for the generation of a pseudorandom integer from the state, *SplitMix* uses the finalization mix of the MurmurHash3¹⁰

¹⁰<https://bit.ly/2tI7IqW>

hash function. This is sufficient as long as γ is not a simple value like 1, even or some other problematic value.

The following implementation (with `state` and `gamma` properly seeded) of 32 bit *SplitMix* was used:

```
uint32_t splitmix32() {
    static uint64_t state = 0xbad0ff1ced15ea5e;
    static uint64_t gamma = 0x9e3779b97f4a7c15
                          | 1;

    uint64_t seed = state;
    state += gamma;
    seed ^= seed >> v;
    seed *= m5;
    seed ^= seed >> w;
    seed *= m6;
    return result_type(seed >> 32);
}
```

The `| 1` after the seed of `gamma` takes care of even gammas which would degrade the quality of the pseudorandom numbers generated.

The *SplitMix* PRNG used for the evaluation—**SplitMix32**—uses the following parameters: `m5` = 0x62a9d9ed799705f5, `m6` = 0xcb24d0a5c88c35b3, `v` = 33 and `w` = 28.

2.2.11 Combinations of different PRNG

The following combined PRNG were compared:

- **DualRand**: LCG with $a = 0x10405$, $c = 0x3035$, $m = 2^{32} - 1$ XORed with a LFSR approximated by $X_n = X_{n-1 \bmod 64} \oplus X_{n-33 \bmod 64} \bmod 2$ on a 128 bit state
- **TripleRand**: *DualRand* XORed with *Hurd288*

2.2.12 Biased Uniform Integer Distribution

The PRNGs listed above return pseudorandom integers in the range of 0 to $2^{32} - 1$ or some other arbitrary range. But for the use in a RANDOM page

2.2 Compared Pseudorandom Number Generators

eviction algorithm, the numbers need to be in the range of the buffer frame indexes.

This requires an algorithm that transforms pseudorandom numbers uniformly distributed in a given range to pseudorandom numbers in the wanted range, keeping the uniform distribution. A blog post¹¹ by Melissa E. O'Neill revealed, that this transformation is the bottleneck of fast random number generation when the `std::uniform_int_distribution` from the *C++ Standard Library* is used.

Therefore, the algorithm that turned out to be the fastest in her comparison was used in the evaluation of PRNGs for RANDOM page evictioners. In contrast to the algorithm built into the *C++ Standard Library*, this one returns pseudorandom numbers in a biased uniform distribution when the range of the used PRNG is not a multiple of the range of the buffer frame indexes. For example, if the PRNG returns numbers uniformly distributed in the integer interval [1..6] and if the buffer frame indexes are in the interval [1..4], this algorithm returns 1 and 2 with a probability of $\frac{1}{3}$ and 3 and 4 with a probability of $\frac{1}{6}$. But as long as the range of buffer frame indexes is much smaller than the range of the used PRNG, the bias is less severe.

For a PRNG returning random numbers in the range of 0 to $2^{32} - 1$, the algorithm is as follows:

```
uint32_t biased_int_dist(uint32_t ranNum,
                        uint32_t rangeMin,
                        uint32_t rangeMax) {
    uint64_t r = uint64_t(rangeMax - rangeMin);
    uint64_t m = uint64_t(ranNum) * (r + 1);
    return uint32_t(rangeMin + (m >> 32));
}
```

The actual implementation used—available on GitHub¹²—works with PRNGs returning integers in any range as well as floating point numbers. It uses metaprogramming to utilize compile-time calculation wherever possible.

¹¹<http://www.pcg-random.org/posts/bounded-rands.html>

¹²<https://bit.ly/37RQQNx>

2.3 Performance Evaluation

Benchmarks of an exemplary DBMS using all the compared RANDOM page evictioners revealed, that there is no statistically significant difference in the hit rates achieved with these different RANDOM page evictioners. Therefore, the only performance difference between the different RANDOM page evictioners is the overhead imposed by the generation of pseudorandom numbers. For this reason, a microbenchmark measuring only the execution time of the PRNGs is appropriate.

The only variable of the evaluation is the PRNG used—the alternatives presented in the previous section are evaluated.

Another potential variable is the number of threads generating pseudorandom numbers. But the behavior of the PRNGs when used concurrently is usually not specified and therefore, all the PRNGs would require synchronization when used by multiple evicting threads. But due to the fact, that the quality of the pseudorandom numbers generated does not matter here, it is assumed, that each evicting thread uses its own thread-local instance of the used PRNG to choose candidates for eviction. And those thread-local instances scale perfectly as long as there are hardware threads available and therefore, an evaluation on one thread is sufficient.

Different algorithms to generate the pseudorandom integers uniformly distributed in a given range could also be compared. But a quick comparison of the custom algorithm presented in Subsection 2.2.12 with the ones provided by the *C++ Standard Library*¹³ and by the *Boost Random Number Library*¹⁴ showed, that the used one¹⁵ is never slower than the competition.

The smallest (> 0) and largest integers returned by the PRNGs—representing the smallest and largest buffer pool indexes—do not significantly influence the performance of the RANDOM page evictioners. Therefore, this integer interval $[1..53467]$ is a constant in this evaluation.

¹³<https://bit.ly/39Xuiwn>

¹⁴<https://bit.ly/37JKHTs>

¹⁵The classic modulo algorithm was used for **rand**, **xorwow** and **xorshift128+**.

2.3 Performance Evaluation

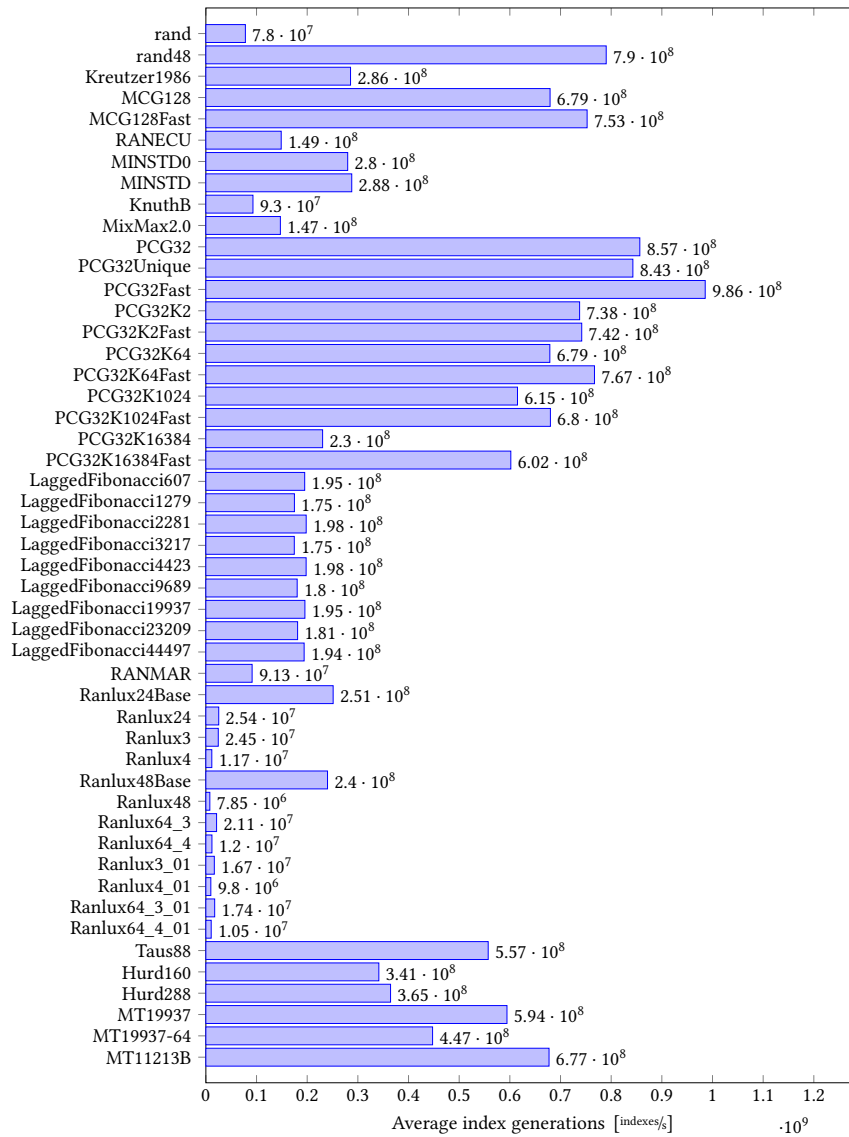


Figure 2.1: The index generation throughput of the evaluated RANDOM implementations (1 of 2)

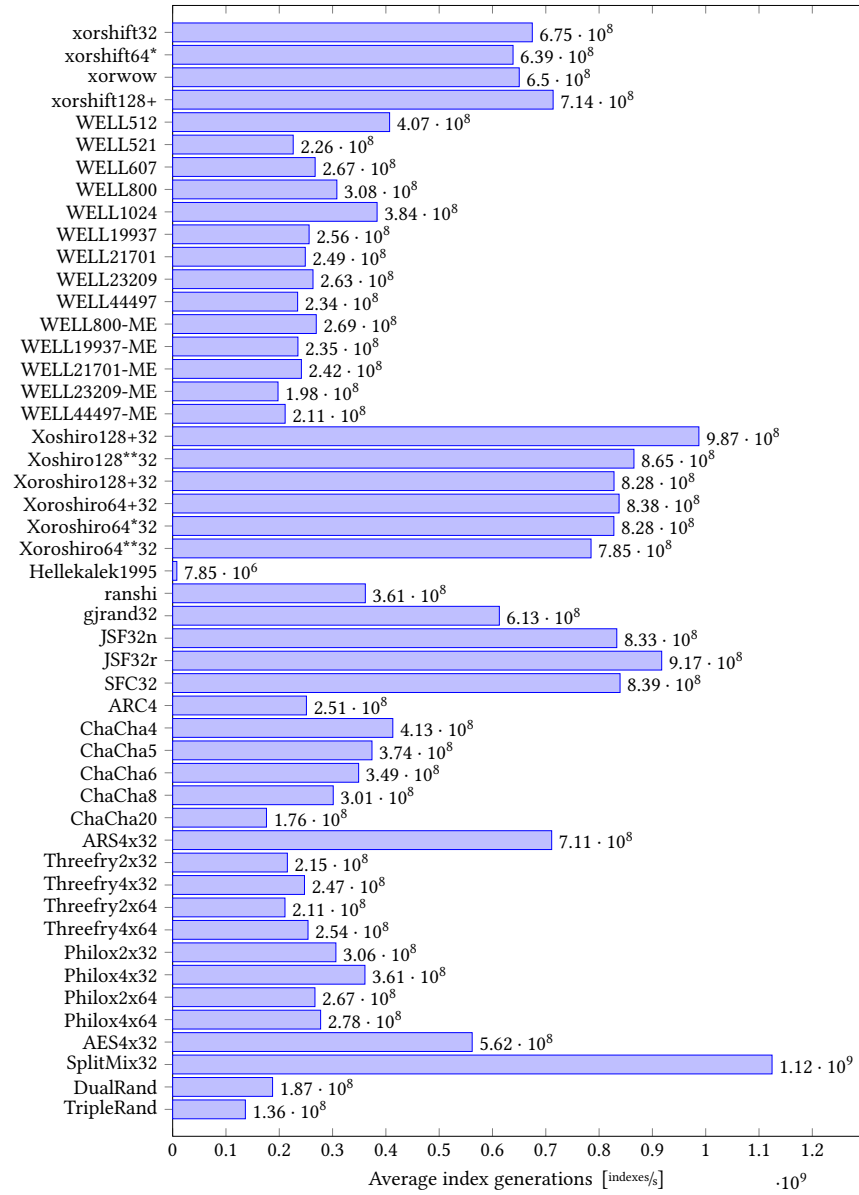


Figure 2.2: The index generation throughput of the evaluated RANDOM implementations (2 of 2)

2.4 Conclusion

2.3.1 Microbenchmark

The microbenchmark used for the performance evaluation of the RANDOM page eviction algorithms instantiates the evaluated PRNG with a seed—generated using `std::random_device`¹⁶—, calculates a given number (5 000 000) of pseudorandom integers in the interval [1..53467] using the PRNG and measures the wall time elapsed.

2.3.2 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

2.3.3 Benchmark Results

Figures 2.1 and 2.2 show the index generation throughput of the evaluated RANDOM page eviction implementations.

The ICG **Hellekalek1995** and the SWBs of the **Ranlux** family (not the non-discarding “base” ones) are the slowest PRNGs in the evaluation. The XOR-based *LSFR* PRNGs and the LCGs of the **PCG** and **MCG** families are among the fastest PRNGs. The very recent **SplitMix32** PRNG described in Subsection 2.2.10 is by far the fastest algorithm in the competition with an average of 1 124 474 870 indexes/s on the used system.

2.4 Conclusion

Like any other DBMS component evaluated for this thesis, the performance (the overhead of the eviction candidate selection, not the achieved hit rate) of the RANDOM eviction algorithm is not critical in most cases. But the hit rate achieved with the chosen page eviction strategy is a major performance factor of a database system.

When it comes to the selection of a page eviction strategy, the RANDOM eviction strategy is the worst but simplest option. The simplicity makes the

¹⁶<https://bit.ly/306x2TE>

RANDOM page eviction strategy a valid choice when it is expected that the main memory is (almost) always large enough to contain the complete working set of a DBS. In this case, the RANDOM eviction strategy would not perform (much) worse than any other “good” page eviction strategy.

When the RANDOM page eviction strategy is chosen for the use in a buffer pool manager of a DBMS, there is basically no reason not to use the fastest PRNG available in the particular programming language. The fastest PRNG in this comparison—*SplitMix*—is part of the Java Development Kit and good implementations are also available in Haskell and C++. The slightly slower XOR-based *LSFR* PRNGs are available for many programming languages and the mid-range PRNG **MT19937** is the default PRNG for most of the programming languages. Most of the fast general-purpose PRNGs can be easily implemented in any programming language typically used in the development of a DBMS and therefore, the lack of such a PRNG in a particular programming language can quickly be compensated.

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 2.2.9, 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 Compared Counter Implementations

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 = 0;
    static uint32_t moduloDivisor = blockCount
                                   - 1;
    return (lastIndex++ % moduloDivisor) + 1;
}
```

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 fact that the eviction decisions of LOOP page evictioners is mostly implementation independent, 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

3.3 Performance Evaluation

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 but when the largest buffer pool index is a power of 2, the modulo operations of the *modulo counter* and *local modulo counter* are significantly faster. Therefore, this integer interval $[1..53467]$ —where 53467 is not a power of 2 because the evaluation should be as general as possible—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

- **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

3.3.3 Microbenchmark Results

Figure 3.1 shows the LOOP index generation throughput which is total number of indexes generated (from all working threads) per time.

The *blocking counters*—*mutex counter* — and *spinlock counter* —are the slowest counters when there are multiple working threads. The locking overhead and—for higher numbers of working threads—the lock contention make these concurrent counters up to 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 *modulo counter* is identical to the *local modulo counter* but the *lock-free counter* suffers from the overhead due to the conditional branch operation. 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*.

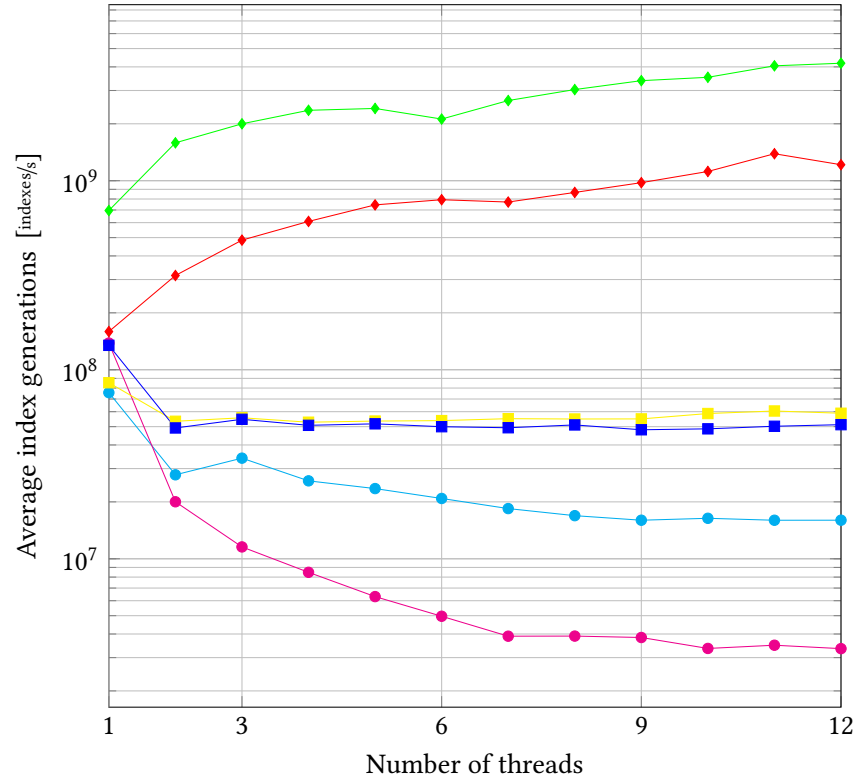




Figure 3.1: The throughput of index generations of the evaluated LOOP implementations

3.4 Conclusion

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/2F9om1S> (visited on Dec. 31, 2019).
- [BD76] Carter Bays and S. D. Durham. “Improving a Poor Random Number Generator”. In: *ACM Transactions on Mathematical Software* 2.1 (Mar. 1976), pp. 59–64. ISSN: 0098-3500. DOI: 10.1145/355666.355670.
- [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.
- [Ber08] Daniel J. Bernstein. *ChaCha, a variant of Salsa20*. Jan. 28, 2008. URL: <https://bit.ly/2raJ7KB> (visited on Dec. 20, 2019).

Bibliography

- [BV18] David Blackman and Sebastiano Vigna. “Scrambled Linear Pseudorandom Number Generators”. In: *Computing Research Repository* (2018). eprint: 1805.01407.
- [Bre92] Richard P. Brent. “Uniform Random Number Generators for Supercomputers”. In: (1992): *Proceedings Fifth Australian Supercomputer Conference, 1949*. URL: <https://bit.ly/2Ph1y9K> (visited on Dec. 12, 2019).
- [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.
- [EL86] Jürgen Eichenauer and Jürgen Lehn. “A Non-Linear Congruential Pseudo Random Number Generator”. In: *Statistische Hefte* 27.1 (Dec. 1986), pp. 315–326. ISSN: 1613-9798. DOI: 10.1007/BF02932576.
- [Gut95] F. Gutbrod. “A Fast Random Number Generator for the Intel Paragon Supercomputer”. In: *Computer Physics Communications* 87.3 (June 1995). Ed. by P. G. Burke et al., pp. 291–306. ISSN: 0010-4655. DOI: 10.1016/0010-4655(95)00005-Z.
- [Har09] Shin Harase. “Maximally Equidistributed Pseudorandom Number Generators via Linear Output Transformations”. In: *Mathematics and Computers in Simulation* 79.5 (Jan. 2009). Ed. by R. Beauwens, pp. 1512–1519. ISSN: 0378-4754. DOI: 10.1016/j.matcom.2008.06.006.

- [Hel95] Peter Hellekalek. “Inversive Pseudorandom Number Generators: Concepts, Results and Links”. In: (Dec. 1995): *Proceedings of the 1995 Winter Simulation Conference*. Ed. by C. Alexopoulos et al., pp. 255–262. DOI: 10.1109/WSC.1995.478732.
- [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).
- [Hur74] William. J. Hurd. “Efficient Generation of Statistically Good Pseudonoise by Linearly Interconnected Shift Registers”. In: *IEEE Transactions on Computers* C-23.2 (Feb. 1974), pp. 146–152. ISSN: 2326-3814. DOI: 10.1109/T-C.1974.223877.
- [Jam90] F. James. “A Review of Pseudorandom Number Generators”. In: *Computer Physics Communications* 60.3 (Oct. 1990). Ed. by P. G. Burke et al., pp. 329–344. ISSN: 0010-4655. DOI: 10.1016/0010-4655(90)90032-V.
- [Knu81] Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Boston, MA, USA: Addison-Wesley, 1981. ISBN: 0-201-03822-6.
- [Kre86] Wolfgang Kreutzer. *System Simulation Programming Styles and Languages*. Boston, MA, USA: Addison-Wesley, 1986. ISBN: 0-201-12914-0.

Bibliography

- [LEc88] P. L’Ecuyer. “Efficient and Portable Combined Random Number Generators”. In: *Communications of the ACM* 31.6 (June 1988). Ed. by Peter J. Denning, pp. 742–751. ISSN: 0001-0782. DOI: 10.1145/62959.62969. URL: <https://bit.ly/34GQTd6> (visited on Dec. 19, 2019).
- [LEc96] Pierre L’Ecuyer. “Maximally Equidistributed Combined Tausworthe Generators”. In: *Mathematics of Computation* 65.213 (Jan. 1996), pp. 203–213. ISSN: 0025-5718. DOI: 10.1090/S0025-5718-96-00696-5. URL: <https://bit.ly/2PHY9EC> (visited on Dec. 13, 2019).
- [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).
- [Leh51] Derrick H. Lehmer. “Mathematical Methods in Large-scale Computing Units”. In: *The Annals of the Computation Laboratory of the Harvard University XXVI (1951): Proceedings of a Second Symposium on Large-Scale Digital Calculating Machinery, 1949*. Ed. by Howard H. Aiken, pp. 141–146. URL: <https://bit.ly/2Pf8c0i> (visited on Dec. 11, 2019).
- [LSS12] Charles E. Leiserson, Tao B. Schardl, and Jim Sukha. “Deterministic Parallel Random-Number Generation for Dynamic-Multithreading Platforms”. In: *SIGPLAN Notices* 47.8 (Aug. 2012). Ed. by Andy Gill, pp. 193–204. ISSN: 0362-1340. DOI: 10.1145/2370036.2145841. URL: <https://bit.ly/2uxe90D> (visited on Jan. 9, 2020).
- [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.

- [Mar03] George Marsaglia. “Xorshift RNGs”. In: *Journal of Statistical Software, Articles* 8.14 (July 4, 2003), pp. 1–6. ISSN: 1548-7660. DOI: 10.18637/jss.v008.i14.
- [MZ91] George Marsaglia and Arif Zaman. “A New Class of Random Number Generators”. In: *The Annals of Applied Probability* 1.3 (Aug. 1991). Ed. by J. Michael Steele, pp. 462–480. ISSN: 2168-8737. DOI: 10.1214/aoap/1177005878.
- [MZT90] George Marsaglia, Arif Zaman, and Wai Wan Tsang. “Toward a Universal Random Number Generator”. In: *Statistics & Probability Letters* 9.1 (Jan. 1990). Ed. by R. A. Johnson, pp. 35–39. ISSN: 0167-7152. DOI: 10.1016/0167-7152(90)90092-L.
- [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).
- [ONe14] Melissa E. O’Neill. *PCG: A Family of Simple Fast Space-Efficient Statistically Good Algorithms for Random Number Generation*. technical report. Harvey Mudd College, Sept. 5, 2014. URL: <https://bit.ly/2E9jpcr> (visited on Dec. 12, 2019).
- [PLM06] François Panneton, Pierre L’Ecuyer, and Makoto Matsumoto. “Improved Long-period Generators Based on Linear Recurrences Modulo 2”. In: *ACM Transactions on Mathematical Soft-*

Bibliography

- ware 32.1 (Mar. 2006). Ed. by Ian Gladwell, pp. 1–16. ISSN: 0098-3500. DOI: 10.1145/1132973.1132974.
- [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).
- [Rot60] A. Rotenberg. “A New Pseudo-Random Number Generator”. In: *Journal of the ACM* 7.1 (Jan. 1960). Ed. by Mario L. Juncosa, pp. 75–77. ISSN: 0004-5411. DOI: 10.1145/321008.321019.
- [Sal+11] John K. Salmon et al. “Parallel Random Numbers: As Easy As 1, 2, 3”. In: (2011): *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*. DOI: 10.1145/2063384.2063405.
- [ST91] G. K. Savvidy and N. G. Ter-Arutyunyan-Savvidy. “On the Monte Carlo Simulation of Physical Systems”. In: *Journal of Computational Physics* 97.2 (Dec. 1991), pp. 566–572. ISSN: 0021-9991. DOI: 10.1016/0021-9991(91)90015-D.
- [SLF14] Guy L. Steele, Doug Lea, and Christine H. Flood. “Fast Splittable Pseudorandom Number Generators”. In: *SIGPLAN Notices* 49.10 (Oct. 2014). Ed. by Andy Gill, pp. 453–472. ISSN: 0362-1340. DOI: 10.1145/2714064.2660195. URL: <https://bit.ly/2t1MB2V> (visited on Jan. 9, 2020).
- [Tau65] Robert C. Tausworthe. “Random Numbers Generated by Linear Recurrence Modulo Two”. In: *Mathematics of Computation* 19.90 (Apr. 1965), pp. 201–209. ISSN: 1088-6842. DOI: 10.1090/S0025-5718-1965-0184406-1.

- [Tho58] W. E. Thomson. “A Modified Congruence Method of Generating Pseudo-random Numbers”. In: *The Computer Journal* 1.2 (Jan. 1958), pp. 83, 86. issn: 0010-4620. DOI: 10.1093/comjnl/1.2.83.
- [Vig17] Sebastiano Vigna. “Further Scramblings of Marsaglia’s Xor-shift Generators”. In: *Journal of Computational and Applied Mathematics* 315 (May 1, 2017), pp. 175–181. issn: 0377-0427. DOI: 10.1016/j.cam.2016.11.006.